

全国高等学校计算机教育研究会教材建设立项项目

微机原理与接口技术

——嵌入式系统描述

姚琳 万亚东 汪红兵 编著

清华大学出版社

微机原理与接口技术

——嵌入式系统描述

姚　琳　万亚东　汪红兵　编著

清华大学出版社
北　京

内 容 简 介

本书内容全面、重点明确、表述简洁,注重将微机接口控制器的基本原理和实际操作相结合,突出软硬件设计中的计算思维模式。全书共 12 章,内容包括微机原理及基本概念、Cortex-M3 处理器体系结构、ARM 汇编、嵌入式系统开发基础、GPIO 控制器、NVIC 及 EXTI 中断控制器、定时器、USART 总线、IIC 总线、SPI 总线、ADC 以及低功耗控制,并配套基于 STM32L15x 系列的实验教程。

本书适合作为非计算机专业微机原理及接口技术的教材,也可作为计算机类嵌入式系统课程的参考教材。

图书在版编目(CIP)数据

微机原理与接口技术:嵌入式系统描述/姚琳,万亚东,汪红兵编著.—北京:清华大学出版社,2019
ISBN 978-7-302-52859-3

Ⅰ.①微…　Ⅱ.①姚…　②万…　③汪…　Ⅲ.①微型计算机-理论　②微型计算机-接口技术　Ⅳ.①TP36

中国版本图书馆 CIP 数据核字(2019)第 082374 号

责任编辑:谢　琛
封面设计:常雪影
责任校对:焦丽丽
责任印制:沈　露

出版发行:清华大学出版社
网　　址:http://www.tup.com.cn, http://www.wqbook.com
地　　址:北京清华大学学研大厦 A 座　　邮　　编:100084
社 总 机:010-62770175　　邮　　购:010-62786544
投稿与读者服务:010-62776969, c-service@tup.tsinghua.edu.cn
质量反馈:010-62772015, zhiliang@tup.tsinghua.edu.cn
课件下载:http://www.tup.com.cn,010-62795954
印 装 者:三河市君旺印务有限公司
经　　销:全国新华书店
开　　本:185mm×260mm　　印　　张:25.5　　字　　数:589 千字
版　　次:2019 年 8 月第 1 版　　印　　次:2019 年 8 月第1 次印刷
定　　价:69.00 元

产品编号:083063-01

前言

随着智能制造、物联网、大数据技术的推进和应用,以及新工科建设的需求,数据的采集和感知成为这些技术应用不可或缺的重要环节,各种物联网大赛、创新创业大赛都对软硬件系统设计能力提出了很高的要求,需要学生具有数据感知、处理、传输和分析的综合能力;此外,随着计算思维在计算机基础教学方面的不断推进,思维能力培养已成为教育教学界的共识,计算机硬件系统结构中包含大量计算思维的知识点,如 RISC、CISC、哈佛体系结构、Cache 分层存储、中断处理及优化机制、流水线、串行并行总线技术等,是计算思维培养非常有效的一门课程。微机原理与接口技术是非计算机专业计算机硬件教育的重要课程,本教材以嵌入式系统为对象,对微机的基本原理、ARM 微处理器的接口技术进行梳理,结合大量实验培养学生计算机硬件素养和计算思维能力,提高学生在计算机软硬件系统设计、调试和创新方面的能力,适用于本科非计算机专业学生。

本书选用 Cortex-M3 处理器内核的 STM32L152 系列低功耗微控制器对 ARM 嵌入式系统的体系结构进行讲述,教材以计算机硬件体系涉及的计算思维为主线,第 1 章阐述微型计算机的基本概念、内部架构和嵌入式系统概念;第 2 章以 ARM Cortex-M3 的处理器工作模式、流水、中断等为案例具体阐述硬件设计的方法;第 3 章介绍汇编指令编码、寻址技术并对启动代码进行了分析;第 4 章简述了嵌入式开发流程及 C 语言基础;第 5 ~ 11 章对常用外围控制器 GPIO、EXTI、Timer、USART、IIC、SPI、ADC 的一般性工作原理、STM32L1 系列处理器的具体实现和特色、寄存器级别和库函数级别两个层次的程序设计方法进行了详细阐述;第 12 章对低功耗设计进行了介绍。教材内容兼顾嵌入式处理器及外围控制器原理讲解和应用程序设计,让读者理解 Cortex-M3 处理器的特性,各种控制器的工作原理及使用方法,理解嵌入式处理器架构。

本教材目标定位为软硬件协同设计思维,而不仅仅是会使用和开发嵌入式系统,结合实验设计,让学生必须理解 ARM 架构、外围控制器的工作原理和设计思路,能够进行应用系统设计。

本书适用于工科非计算机专业微机接口技术、嵌入式系统课程,也可作为计算机专业嵌入式开发课程的教材。

作　者

2019 年 4 月

目录

Contents

第 1 章　微型计算机与嵌入式系统概论 …… 1

1.1　微型计算机概述 …… 1

1.1.1　微型计算机系统的组成 …… 1

1.1.2　微处理器的发展 …… 3

1.2　微型计算机的基本原理 …… 7

1.2.1　冯·诺依曼体系结构 …… 7

1.2.2　微机的总线 …… 8

1.2.3　哈佛体系结构 …… 14

1.2.4　微处理器的内部结构 …… 15

1.2.5　I/O 接口技术 …… 19

1.2.6　存储器 …… 20

1.2.7　程序的执行过程 …… 24

1.3　嵌入式系统概述 …… 25

1.4　嵌入式系统架构 …… 27

1.5　嵌入式系统的典型应用 …… 28

1.6　典型嵌入式开源硬件和软件系统 …… 31

1.6.1　开源硬件平台 …… 31

1.6.2　嵌入式开源操作系统 …… 33

第 2 章　Cortex-M3 微处理器的体系结构 …… 35

2.1　ARM 微处理器系列介绍 …… 35

2.2　ARM Cortex-M3 体系结构 …… 37

2.2.1　总体架构 …… 37

2.2.2　操作模式 …… 39

2.2.3　寄存器 …… 40

2.2.4　总线 …… 44

2.2.5　存储器 …… 45

2.2.6　中断 …… 50

2.3 STM32L152RET6 微处理器介绍 …… 50
2.4 STM32L152RET6 微处理器的系统结构 …… 51
2.5 STM32L152RET6 微处理器的引脚说明 …… 54
2.6 STM32L152RET6 微处理器的复位和时钟控制 …… 57
2.7 STM32L152RET6 微处理器的存储映射 …… 59

第 3 章 Cortex-M3 处理器的指令系统 …… 63

3.1 Cortex-M3 处理器的指令系统概述 …… 63
3.1.1 指令系统基本概念 …… 63
3.1.2 指令格式 …… 65
3.1.3 寻址方式 …… 67
3.1.4 数据传送指令 …… 68
3.1.5 存储器访问指令 …… 69
3.1.6 算术运算指令 …… 74
3.1.7 逻辑运算指令 …… 77
3.1.8 移位和循环指令 …… 78
3.1.9 比较指令 …… 79
3.1.10 分支控制指令 …… 80
3.1.11 其他指令 …… 81
3.2 ARM 汇编器中的伪指令 …… 83
3.2.1 Thumb 伪指令 …… 83
3.2.2 符号定义伪指令 …… 84
3.2.3 数据定义伪指令 …… 85
3.2.4 汇编控制伪指令 …… 86
3.2.5 其他常用的伪指令 …… 87
3.3 汇编语言的程序结构 …… 88

第 4 章 开发板硬件系统及开发环境 …… 92

4.1 最小系统设计 …… 92
4.2 开发板电路原理图 …… 93
4.2.1 电源 …… 93
4.2.2 复位和启动电路 …… 95
4.2.3 时钟 …… 95
4.2.4 调试接口 …… 96
4.2.5 按键 …… 97
4.2.6 LED 灯 …… 97
4.2.7 显示屏 …… 98
4.2.8 扩展 I/O 口 …… 98

4.3 软件开发环境 …… 99
4.3.1 嵌入式软件开发流程 …… 99
4.3.2 程序开发库 CMSIS …… 101
4.3.3 STM32L52 嵌入式程序开发预备知识 …… 103

第 5 章 通用输入输出 …… 112

5.1 GPIO 原理 …… 112
5.1.1 GPIO 功能 …… 112
5.1.2 I/O 模式配置 …… 113
5.2 GPIO 寄存器 …… 115
5.3 GPIO 操作函数库 …… 120
5.4 GPIO 实例 …… 128
5.4.1 GPIO 寄存器基本操作 …… 128
5.4.2 GPIO LED 灯控制 …… 130
5.4.3 GPIO 按键输入 …… 131

第 6 章 异常和中断处理技术 …… 133

6.1 中断的基本概念 …… 133
6.2 中断向量表 …… 134
6.3 中断的执行过程 …… 138
6.3.1 中断响应基本流程 …… 138
6.3.2 中断优化技术 …… 141
6.3.3 系统异常 …… 142
6.4 嵌套向量中断控制器 NVIC …… 144
6.4.1 STM32L152 NVIC …… 144
6.4.2 NVIC 寄存器 …… 144
6.4.3 系统异常处理 …… 149
6.4.4 全局中断管理 …… 150
6.4.5 NVIC 库函数 …… 151
6.5 外部中断/事件控制器 EXTI …… 157
6.6 寄存器说明 …… 158
6.7 EXTI 函数库 …… 162
6.8 中断案例 …… 164

第 7 章 定时器 …… 168

7.1 定时器原理概述 …… 168
7.2 内部定时器 SysTick …… 170
7.2.1 SysTick 寄存器 …… 170

7.2.2 SysTick 定时器库函数 …… 172
7.2.3 SysTick 定时器应用例程 …… 174
7.3 外围定时器基本概念 …… 175
7.4 基本定时器 TIM6、TIM7 …… 179
7.5 通用定时器 TIM2 ~ TIM4、TIM9 ~ TIM11 …… 181
7.5.1 通用定时器时基单元 …… 182
7.5.2 通用定时器输入捕获和输出比较单元 …… 185
7.5.3 TIMx 的外部触发同步模式 …… 189
7.6 定时器寄存器 …… 191
7.7 外围定时器库函数 …… 197
7.8 定时器应用例程 …… 205
7.8.1 定时器寄存器操作案例 …… 205
7.8.2 基本计时中断示例 …… 206
7.8.3 比较输出示例 …… 208
7.8.4 输入捕获示例 …… 212
7.8.5 PWM 输出和输入示例 …… 215

第 8 章 USART 串口控制器 …… 218

8.1 串行输入输出接口的基本概念 …… 218
8.2 串行通信协议 …… 219
8.2.1 异步串行通信协议 …… 219
8.2.2 同步串行通信协议 …… 220
8.2.3 串行通信基本概念 …… 221
8.3 STM32L152 USART 内部结构与原理 …… 224
8.3.1 发送器 …… 226
8.3.2 接收器 …… 228
8.3.3 校验控制 …… 232
8.3.4 硬件流控制 …… 232
8.3.5 USART 中断请求 …… 233
8.4 USART 寄存器 …… 234
8.5 USART 数据传输配置 …… 240
8.5.1 波特率计算 …… 240
8.5.2 异步双向通信模式配置 …… 241
8.6 USART 帧传输协议 …… 242
8.6.1 串行链路帧格式设计 …… 242
8.6.2 MODBUS 帧格式 …… 246
8.7 USART 函数库 …… 247
8.7.1 寄存器定义 …… 247

8.7.2 USART 库函数 …… 250
8.8 USART 案例 …… 257
8.8.1 串口寄存器操作案例 …… 257
8.8.2 串口配置基本流程 …… 258
8.8.3 PC 串口通信案例 …… 259
8.8.4 状态机多字节数据帧发送和接收案例 …… 261

第 9 章 IIC 总线 …… 267

9.1 IIC 总线概述 …… 267
9.2 I2C 总线的基本操作 …… 268
9.3 STM32L152 I2C 总线控制器 …… 273
9.4 I2C 寄存器描述 …… 275
9.5 I2C 数据通信流程 …… 281
9.5.1 I2C 从模式通信 …… 281
9.5.2 I2C 主模式通信 …… 283
9.5.3 总线通信错误 …… 285
9.5.4 中断请求 …… 285
9.6 函数库 …… 287
9.6.1 I2C 寄存器结构 …… 287
9.6.2 I2C 库函数 …… 288
9.7 I2C 案例 …… 298
9.7.1 I2C 寄存器操作案例 …… 298
9.7.2 I2C 基本配置 …… 299
9.7.3 模拟 I2C 实现 …… 301
9.7.4 串行 Flash 通信 …… 304
9.7.5 ADT7420 温度传感器通信 …… 306

第 10 章 SPI …… 309

10.1 SPI 总线概述 …… 309
10.2 SPI 总线控制器架构 …… 310
10.2.1 接口信号和连接方式 …… 310
10.2.2 传输模式和时序 …… 313
10.2.3 STM32L15x SPI 总线控制器 …… 315
10.3 SPI 寄存器说明 …… 317
10.4 SPI 通信流程 …… 320
10.4.1 SPI 双工通信模式配置 …… 321
10.4.2 SPI 单工/半双工通信 …… 323
10.5 函数库 …… 326

10.5.1 SPI 寄存器结构 …… 326
10.5.2 SPI 库函数 …… 328
10.6 SPI 案例 …… 333
10.6.1 SPI 寄存器操作案例 …… 333
10.6.2 SPI 函数库案例 …… 334
10.6.3 温度传感器 ADT7320 案例 …… 335

第 11 章 模拟/数字转换 …… 339

11.1 ADC 简介 …… 339
11.2 STM32L152 ADC …… 344
11.2.1 STM32L152 ADC 功能 …… 346
11.2.2 温度和电压转换 …… 357
11.3 ADC 寄存器 …… 358
11.4 ADC 寄存器结构及 ADC 库函数 …… 365
11.4.1 ADC 寄存器结构 …… 366
11.4.2 ADC 库函数 …… 368
11.5 ADC 案例 …… 377
11.5.1 ADC 寄存器操作案例 …… 377
11.5.2 ADC 库函数操作案例 …… 378

第 12 章 低功耗技术 …… 380

12.1 处理器功耗的构成/类型 …… 380
12.1.1 动态功耗 …… 380
12.1.2 静态功耗 …… 381
12.2 STM32L1 系列处理器低功耗设计 …… 382
12.2.1 STM32 的电源系统 …… 382
12.2.2 动态电压调节管理 …… 383
12.2.3 电源检测 …… 385
12.2.4 低功耗模式 …… 386
12.3 功耗控制寄存器 …… 389
12.4 PWR 寄存器结构及库函数 …… 391
12.4.1 PWR 寄存器结构 …… 391
12.4.2 PWR 库函数 …… 391
12.5 PWR 案例 …… 394

参考文献 …… 397

Chapter 1

第1章　微型计算机与嵌入式系统概论

【导读】 嵌入式系统属于微型计算机范畴,本章首先介绍微型计算机的基本组成,核心部件微处理器的发展历史,然后对微处理器工作原理的一些基本概念:组成架构、总线、输入输出、存储系统等进行了介绍。微处理器应用于微机,属于通用计算机系统;而与应用场景结合,形成专用计算系统,称为嵌入式系统,本章对嵌入式系统的概念、组成和典型应用进行介绍,并列举了目前典型的开源嵌入式开发硬件和软件平台。通过本章学习,建立微机系统的整体构成和微处理器设计的相关计算思维方法,对嵌入式系统的概念及软硬件系统有总体的认识。

1.1　微型计算机概述

微型计算机是针对小型计算机、大型计算机和超级计算机而言的,是根据规模和性能进行计算机分类的。一般来说,微型计算机是一种小型的、相对便宜的、以微处理器作为 CPU 的计算机。这类计算机由印制电路板、微处理器、存储器和输入输出电路组成,占用很少的物理空间。随着集成电路技术的发展,微型计算机在 20 世纪 80 年代开始流行,获得广泛的应用。目前我们使用的个人计算机(台式机、笔记本计算机、平板计算机、智能手机、计算器等)、家用娱乐设施(游戏机、智能电视、智能音箱、电子书阅读器等)以及路由器、交换机等通信设备均是微型计算机系统。

1.1.1　微型计算机系统的组成

一个完整的微型计算机系统由硬件系统和软件系统两部分组成, 如图 1-1 所示。

计算机硬件部分包括中央处理器(CPU)、存储器、输入和输出设备。

(1) 微型计算机的中央处理器也称为微处理器(Micro Processor Unit,MPU)。计算机利用 CPU 处理数据,利用存储器存储数据。CPU 是计算机硬件的核心,主要包括运算器和控制器两大部分,控制着整个计算机系统的工作。计算机的性能主要取决于 CPU 的性能。

运算器又称为算术逻辑单元(Arithmetic Logic Unit,ALU),控制器的主要作用是使整个计算机系统能够自动运行。控制器从存储器取出数据,运算器进行算术运算或逻辑运算,

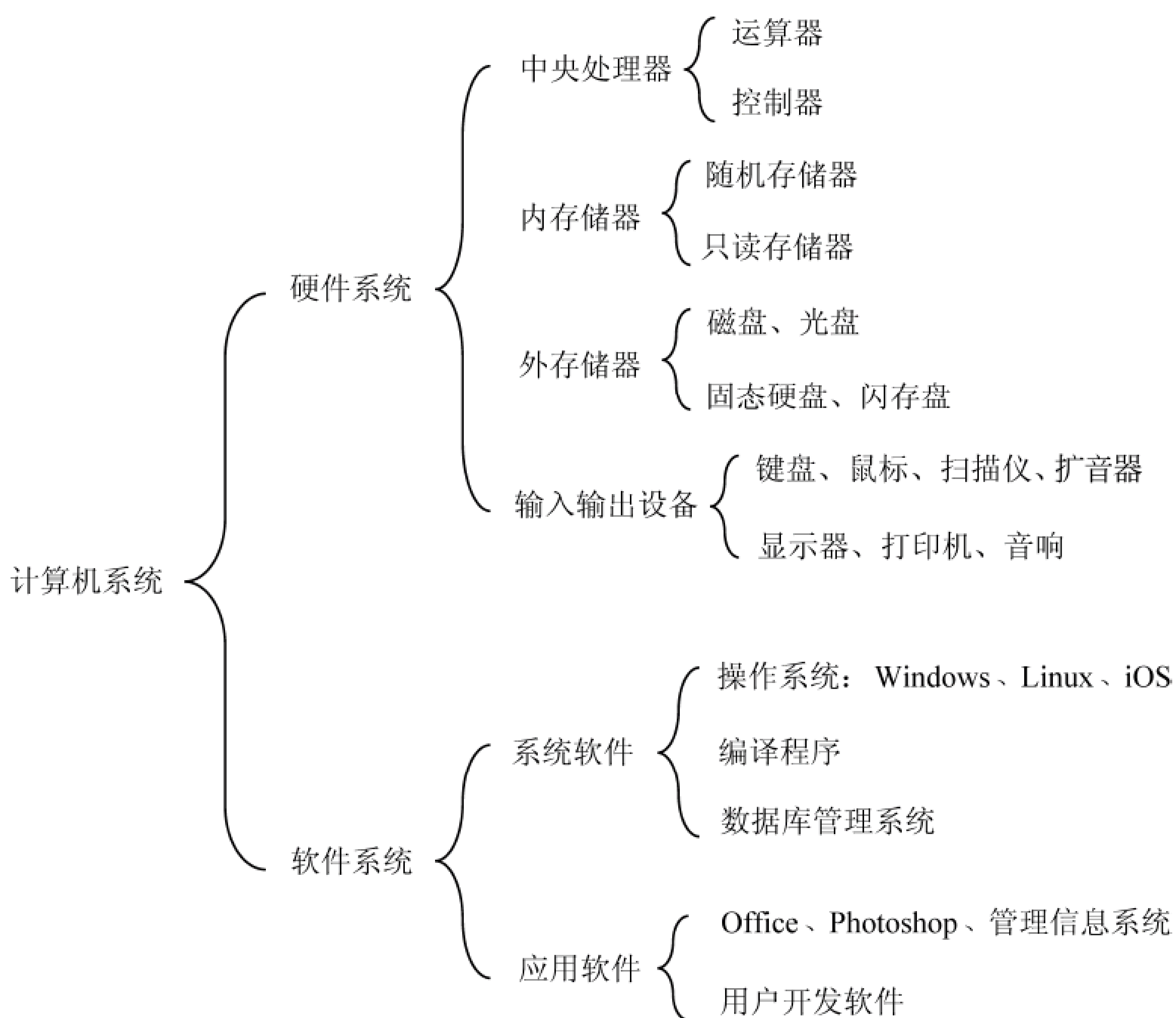


图 1-1　计算机的系统组成

并把处理后的结果送回存储器。执行程序时，控制器从主存中取出相应的指令和数据，然后向其他功能部件发出指令所需的控制信号，完成相应的操作，再从主存中取出下一条指令执行，如此循环，直到程序结束。

(2) 存储器是计算机中的存储部件。存储器分为主存(内存储器)和辅存(外存储器)两大类。在计算机系统中，习惯上把内存、CPU 合称为主机。内存储器分为随机读写存储器(RAM)、只读存储器(ROM)和高速缓冲存储器(Cache)三类。通常生活中的内存一般指的是 RAM。外存储器主要包括硬盘、光盘、U 盘和移动硬盘等。

(3) 输入输出设备主要包括键盘、鼠标、显示器和打印机等。

硬件是组成计算机的基础，软件是计算机的灵魂。计算机的硬件系统上只有安装了软件后，才能发挥其应有的作用，使用不同的软件，计算机可以完成各种不同的工作。微型计算机系统的软件分为两大类，即系统软件和应用软件。

系统软件是指为管理计算机系统的硬件和支持应用软件运行而提供的基本软件，最常用的有操作系统、程序设计语言编译器、数据库管理系统、联网及通信软件等。操作系统是微机最基本、最重要的系统软件，它负责管理计算机系统的各种硬件资源(例如 CPU、内存空间、磁盘空间、外部设备等)，并且负责将用户对机器的管理命令转换为机器内部的实际操作。典型的操作系统有 Linux、Mac OS、Windows 7、Windows 10、iOS 等。

应用软件是指除了系统软件以外，利用计算机为解决某类问题而设计的程序的集合，主要包括信息管理软件、辅助设计软件、实时控制软件等。

1.1.2　微处理器的发展

微型计算机系统的核心是微处理器，微处理器的发展直接影响着微型计算机系统的应用。

1.1.2.1　微处理器发展史

1）早期微处理器

第一款微处理器是美国军方研制的中央空气数据计算机（Center Air Data Computer，CADC），由6颗晶片组成，用于F-14雄猫战机的大气数据测量与控制。1971年，Intel公司发布的4004是世界上第一款商用处理器，主频108kHz，4004和Intel开发出的4001（动态内存DRAM）、4002（只读存储器ROM）、4003（寄存器Register）可架构出一台微型计算机硬件系统。

2）8位微处理器时期

1972年，Intel公司推出的8008微处理器是第一款8位处理器，主频0.5MHz。1974年，Intel公司推出8080处理器，主频2MHz，16位地址总线、8位数据总线，内部集成7个8位寄存器，支持16位内存，同时也包含了一些输入输出端口。

此时，微处理器的优势已被业界所认同，更多公司开始进入微处理器设计，仙童、AMD、摩托罗拉以及Zilog等公司均开始研发微处理器，摩托罗拉1974年发布了MC6800，工作主频1MHz，1976年Zilog公司发布了Z80，性能比8080更强大。

3）16位微处理器时期

1978年，Intel公司首次生产出16位的微处理器，命名为8086，同时还生产出与之相配合的数学协处理器8087，这两种芯片使用相互兼容的指令集，即后来PC使用的X86指令集。

1979年，Intel公司推出了8088芯片，主频4.77MHz，地址总线为20位，寻址范围1MB内存。8088内部数据总线都是16位，外部数据总线是8位（8086是16位）。1981年8088芯片首次用于IBM PC中，开创了全新的微机时代。

1979年，Zilog发布了其第一款16位处理器Z8000，摩托罗拉发布16位处理器MC68000（16位计算单元，32位数据总线）。

1982年，Intel公司推出80286芯片，它比8086和8088都有了飞跃式的发展，虽然它仍旧是16位结构，时钟频率20MHz。其内部和外部数据总线皆为16位，地址总线24位，可寻址16MB内存。

4）32位微处理器时期

世界上第一块单片32位微处理器是1982年AT&T（贝尔）实验室的BELLMAC-32A。

1985年，Intel公司发布了80386，首次在X86处理器中实现了32位系统，集成80387数字辅助处理器增强浮点运算能力，首次采用外置的高速缓存（Cache）解决内存速度瓶颈问题。工作频率也从12.5MHz逐步提高到20MHz、25MHz、33MHz，直至最后的40MHz。



图 1-2　Intel 早期 4 位、8 位、16 位处理器

1985 年摩托罗拉推出了 MC68020，增加了 32 位数据和地址总线，在 UNIX 超级微机市场上获得巨大成功，后续又生产了 MC68030(集成了内存管理)、MC68040(集成浮点运算器)。

1986 年 MIPS 推出 R2000 处理器，1988 年推出 R3000 处理器，采用精简指令集(RISC)设计，成为 RISC 微处理器的代表。

1987 年 Sun 公司推出了第一款 32 位的 SPARC 86900 Sunrise 处理器，这款处理器采用 SPARC V7 架构，采用 0.8μm 工艺，主频 16MHz，主要用于 SUN 工作站(Solaris 系统)。

1989 年 Intel 公司发布 80486，支持虚拟存储管理技术，虚拟存储空间 64TB(支持 48 位的有效虚拟地址)。片内集成有浮点运算部件和 8KB 的 Cache，同时也支持外部 Cache。整数处理部件采用精简指令集 RISC 结构，提高了指令的执行速度。此外，80486 微处理器还引进了时钟倍频技术和新的内部总线结构，从而使主频可以超出 100MHz。

之后，Intel 公司陆续发布了 Pentium 系列处理器 Pentium(超标量)、Pentium Pro(动态执行)、Pentium Ⅱ(MMX 指令集)、Pentium Ⅲ(SIMD)、Pentium 4、Pentium M(低功耗)、Pentium D(双核)、酷睿 Core 等一系列处理器。目前，酷睿系列(Core M、Core I3、Core I5、Core I7、Core I9)已经经过了 8 代的发展。

1991 年，ARM 发布了自己的第一款 RISC 处理器核 ARM6，1993 年推出 ARM7，一直到目前的 ARM11 和 Cortex，在嵌入式微处理器市场占据了大量份额。

5) 64 位微处理器时期

1991 年，MIPS 推出第一款 64 位商用微处理器 R4000，之后又陆续推出 R8000、R10000 和 R12000 等型号，2007 年，中科院计算所龙芯获得 MIPS 处理器 32 位和 64 位的授权。

1992 年，DEC 发布了 64 位的 Alpha 处理器，主要用于工作站和服务器，后 DEC 将 Alpha 技术出售给康柏公司，最终康柏公司将 Alpha 的技术出售给 Intel 公司。

1995 年，Sun 公司推出了 64 位 UltraSPARC Ⅰ微处理器。UltraSPARC I 革新了微处理器的可扩展性和带宽等工业标准，其频率达 143MHz，采用 0.5μm 工艺技术。

2001 年，Intel 公司推出了 IA64 架构的 Itanium 系列处理器，其指令系统与 X86 不兼容，用于服务器市场。

2003 年，AMD 提出了 X86 的 64 位扩展指令集 AMD64，用于服务器市场的 64 位处理器进入到 PC 领域，后来 Intel 公司最终采用了 AMD64。

ARM 于 2011 年发布了 ARMv8 64 位架构，ARMv8 使用了两种执行模式：AArch32 和 AArch64，处理器在运行中可以无缝地在两种模式间切换。这意味着 64 位指令的解码

器是全新设计的，不用兼顾32位指令，而处理器依然可以向后兼容。

1.1.2.2　微控制器发展史

微控制器(Micro Controller Unit，MCU)是一种采用超大规模集成电路技术把具有数据处理能力的微处理器MPU、随机存储器RAM、只读存储器ROM、多种I/O接口和中断系统、定时器/计数器等功能集成到一块硅片上构成的一个小而完善的微型计算机系统。

早期的微控制器称作单片机(Single Chip Microcomputer，SCM)，主要实现单片系统集成，随着嵌入式应用的扩展，系统各种外围接口电路日趋复杂，系统的智能化控制能力凸显，各种电气、控制和电子技术厂家开始进入微处理器的设计和生产，结合SoC(System on Chip)技术，单片机进入微控制器阶段。

1976年Intel公司推出了第一款单片机MCS-48。此后，GI推出了PIC1x系列8位单片机，摩托罗拉推出了MC6800处理器为核心的M6800系列单片机，Zilog推出了Z8系列单片机。

1990年，Intel公司推出了MCS-51，MCS-51采用经典的8位单片机的总线结构，包括8位数据总线、16位地址总线、控制总线及具有很多机通信功能的串行通信接口，并授权给其他厂家使用，Intel、Atmel、Philips和STC等推出了一系列MCS-51核心的单片机，从此成为8位单片机的主流，直到现在还在大量应用。此外，8位单片机的主流架构还有Atmel公司的AVR系列，凌阳科技的SPMC65系列等。

8位单片机占据了较大的市场，16位单片机由于8位单片机的市场地位和32位单片机的快速发展，相对发展空间较小，典型的芯片包括Intel的MCS-96系列单片机，TI的MSP430系列，摩托罗拉的68HC12/16系列以及MicoChip的PIC24系列。

随着嵌入式系统应用的爆发，单片机的运算处理性能和外设通信能力等无法满足复杂应用的计算需求，由此催生了32位微控制器。32位微控制器多采用RISC指令集，典型的代表有MIPS、ARM、摩托罗拉的68K系列等。目前，微控制器基本已被ARM占据，随着物联网应用的发展，芯片厂家的主要在低功耗、低电压、大容量存储和高性能计算上对微控制器进行优化。

1.1.2.3　CISC和RISC设计方法

按照指令系统分类，计算机大致可以分为两类：复杂指令系统计算机(Complex Instruction Set Computer，CISC)和精简指令系统计算机(Reduced Instruction Set Computer，RISC)。CISC是CPU的传统设计模式，其指令系统的特点是指令数目多而复杂，每条指令的长度不尽相等；而RISC则是CPU的一种新型设计模式，其指令系统的主要特点是指令条数少且简单，指令长度固定。

1) CISC指令集

计算机的指令系统最初只有很少的一些基本指令，而其他的复杂指令全靠软件编译时通过简单指令的组合来实现。后来，越来越多的复杂指令被加入到了指令系统中，可用硬件实现复杂的运算。但是，一个指令系统的指令条数受到指令操作码位数的限制，如果操作码

为 8 位,那么指令条数最多为 256 条,而指令的宽度则是很难增加的。操作码扩展可以解决这个问题,在指令格式中,操作码后面跟的是地址码,而有些指令是用不到地址码或只用少量位数的地址码的,那么就可以把操作码扩展到地址码的位置,使操作码的位数得以增加。例如,一个指令系统的操作码为 2 位,那么可以有 00、01、10、11 四条不同的指令。现在把 11 作为保留,把操作码扩展到 4 位,那么就可以有 00、01、10、1100、1101、1110、1111 七条指令,其中 1100、1101、1110、1111 这四条指令的地址码部分必须减少两位。为了达到减少地址码这一操作码扩展的先决条件,设计者提出了各种各样的寻址方式,如基址寻址、相对寻址等,以最大限度地压缩地址码长度,为操作码留出空间。

由此,大量的复杂指令、可变的指令长度、多种寻址方式形成了 CISC 指令集。CISC 指令集的复杂性大大增加了译码的难度,早期计算机运算能力差,译码相对时间较短,现代计算机运算速度大大提升,导致译码上所浪费的时间过长,严重影响了计算机性能。

2) RISC 指令集

1975 年,IBM 对 IBM 370 CISC 系统的研究发现,发现其中仅占总指令数 20%的简单指令却在程序调用中占据了 80%,而占指令数 80%的复杂指令却只有 20%的机会被调用到,即符合经济学中的 80/20 法则,由此提出了 RISC 的概念。20 世纪 80 年代末开始,各家公司的 RISC CPU 大量出现,其中典型代表为 MIPS 和 ARM。

RISC 体系结构的基本思想:针对 CISC 指令系统指令种类太多、指令格式不规范、寻址方式太多的缺点,通过减少指令种类、规范指令格式、简化寻址方式,方便处理器内部的并行处理,提高处理器内部器件的使用效率,从而大幅度地提高处理器的性能。

RISC 的目标决不是简单地缩减指令系统,而是使处理器的结构更简单、更合理,具有更高的性能和执行效率,同时降低处理器的开发成本。由于 RISC 指令系统仅包含最常用的简单指令,因此,RISC 技术可以通过硬件优化设计,把时钟频率提得很高,从而实现整个系统的高性能。同时,RISC 技术在 CPU 芯片上设置大量寄存器,用来把常用的数据保存在这些寄存器中,大大减少对存储器的访问,用高速的寄存器访问取代低速的存储器访问,从而提高系统整体性能。

RISC 的典型特征包括:

(1) 指令种类少,指令格式规范:RISC 指令集通常只使用一种或少数几种格式,指令长度单一(一般 4 字节),并且在字边界上对齐,字段位置(特别是操作码的位置)固定。

(2) 寻址方式简化:几乎所有指令都使用寄存器寻址方式,其他更为复杂的寻址方式,如间接寻址等,则由软件利用简单的寻址方式来合成。

(3) 大量利用寄存器间操作:RISC 指令集中大多数操作都是寄存器到寄存器的操作,只有取数指令、存数指令访问存储器。

(4) 简化处理器结构:使用 RISC 指令集,可以大大简化处理器中的控制器和其他功能单元的设计,不必使用大量专用寄存器,特别是允许以硬连线方式来实现指令操作,以期更快的执行速度,而不必像 CISC 处理器那样使用微程序来实现指令操作。因此,RISC 处理器不必像 CISC 处理器那样设置微程序控制存储器,从而能够快速地直接执行指令。

(5) 加强处理器的并行能力：RISC 指令集非常适合于采用流水线、超流水线和超标量技术，从而实现指令级并行操作，提高处理器的性能。目前常用的处理器的内部并行操作技术，基本上都是基于 RISC 体系结构而逐步发展和走向成熟的。

(6) RISC 技术的复杂性在于它的优化编译程序，因此软件系统开发时间比 CISC 机器要长。

RISC 与 CISC 的主要特征对比如表 1-1 所示。

表 1-1　RISC 与 CISC 的主要特征对比

比 较 项 目	CISC	RISC
指令系统	复杂、庞大	简单、精简
指令数目	一般大于 200	一般小于 100
指令格式	一般大于 4 种	一般小于 4 种
寻址方式	一般大于 4 种	一般小于 4 种
指令字长	不固定	定长
指令执行时间	慢	快
程序代码长度	短	长

1.2　微型计算机的基本原理

1.2.1　冯・诺依曼体系结构

现代计算机基本沿用冯・诺依曼体系结构，其基本设计思想包括以下三点。

1) 计算机系统的组成

运算器、存储器(主存)、控制器、输入设备和输出设备五大部件组成一个完整的计算机系统，如图 1-3 所示。

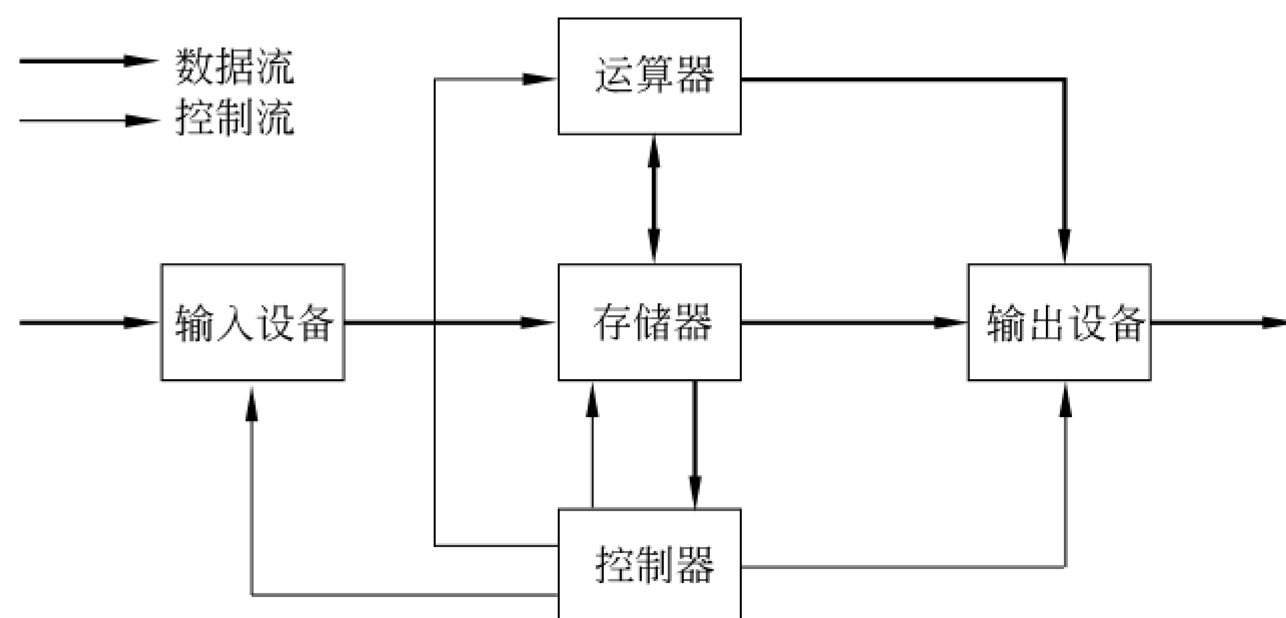


图 1-3　冯・诺依曼体系结构五大功能部件

2）采用二进制形式表示数据和指令

数据和指令都是以二进制形式存储在存储器中，从存储器存储的内容来看两者并无区别，都是由 0 和 1 组成的代码序列，只是各自约定的含义不同。计算机在读取指令时，把从计算机读到的信息看作指令；而在读取数据时，把从计算机读到的信息看作操作数。数据和指令在软件编制中就已加以区分，所以正常情况下两者不会产生混乱。我们通常把存储在存储器中的数据和指令统称为数据，因为程序本身也可以作为被处理的对象进行加工处理，例如对照程序进行编译，就是将源程序当作被加工处理的对象。

3）采用存储程序方式

事先编制程序，将程序（包含指令和数据）存入主存储器中，计算机在运行程序时就能自动地、连续地从存储器中依次取出指令且执行。这是计算机能高速自动运行的基础。计算机的工作体现为执行程序，计算机功能的扩展在很大程度上也体现为所存储程序的扩展。

冯·诺依曼机的这种工作方式，可称为指令流驱动方式，即按照指令的执行序列，依次读取指令，然后根据指令所含的控制信息，调用数据进行处理。因此，在执行程序的过程中，始终以控制信息流为驱动工作的因素，而数据信息流则是被动地被调用处理。为了控制指令序列的执行顺序，设置一个程序（指令）计数器 PC（Program Counter），让它存放当前指令所在的存储单元的地址。如果程序现在是顺序执行的，每取出一条指令后 PC 内容加 1，指示下一条指令该从何处取得。如果程序将转移到某处，就将转移的目标地址送入 PC，以便按新地址读取后继指令。所以，PC 就像一个指针，一直指示着程序的执行进程，也就是指示控制流的形成。由于多数情况下程序是顺序执行的，所以大多数指令需要依次地紧挨着存放，除了个别即将使用的数据可以紧挨着指令存放外、一般将指令和数据分别存放在该程序区的不同区域内。

冯·诺依曼型计算机从本质上讲是采取串行顺序处理的工作机制，即使有关数据已经准备好，也必须逐条执行指令序列。而提高计算机性能的根本方向之一是并行处理。因此，近年来人们谋求突破传统冯·诺依曼体制的束缚，这种努力被称为非诺依曼化，主要表现在以下三个方面。

- 在冯·诺依曼体制范畴内，对传统冯·诺依曼机进行改造，如采用多个处理部件形成流水处理，依靠时间上的重叠提高处理效率；又如组成阵列机结构，形成单指令流多数据流，提高处理速度。
- 用多个冯·诺依曼机组成多机系统，支持并行算法结构。
- 从根本上改变冯·诺依曼机的控制流驱动方式。例如，采用数据流驱动工作方式的数据流计算机，只要数据已经准备好，有关的指令就可并行执行。这是真正非诺依曼化的计算机，它为并行处理开辟了新的前景，但由于控制的复杂性，仍处于实验探索之中。

1.2.2 微机的总线

1.2.2.1 总线基本概念

冯·诺依曼机的五大功能部件之间是通过总线（Bus）进行连接的。总线是用于连接多

个设备的数据通道，是计算机各种功能部件之间传送信息的公共通信干线，它是由导线组成的传输线束，一根线路在同一时间内仅能传输一比特，因此，必须同时采用多条线路才能传送更多数据，总线可同时传输的数据数就称为总线宽度（Bus Width），以比特为单位，总线宽度愈大，传输性能就越高。总线频率是总线工作速度的一个重要参数，总线频率是指一秒钟传输数据的次数，工作频率越高，速度越快。总线频率通常用 MHz 表示，如 33MHz、100MHz、400MHz、800MHz 等。总线的带宽（Bandwidth），即单位时间内可以传输的总数据数，指的是总线本身所能达到的最高数据传输速率。总线带宽＝频率×宽度，单位为 B/s。

总线上传送的信息包括数据信息、地址信息和控制信息，因此，总线按功能分三种：数据总线（Data Bus，DB）、地址总线（Address Bus，AB）和控制总线（Control Bus，CB），如图 1-4 所示。

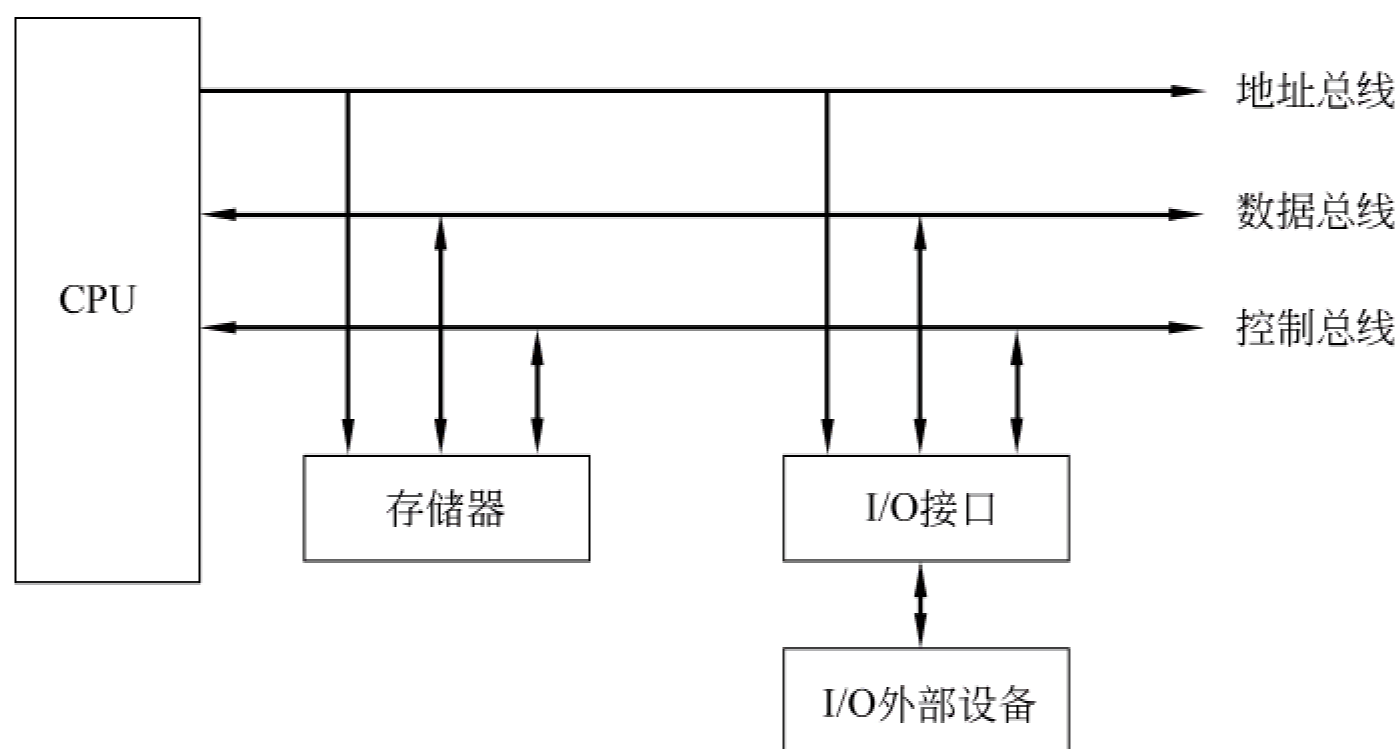


图 1-4　微机系统总线

数据总线用于传送数据信息（包括指令和数据）。数据总线是双向三态总线，既可以把 CPU 的数据信息传送到存储器或 I/O 接口等其他部件，也可以将其他部件的数据信息传送到 CPU。数据总线的位数是微型计算机的一个重要指标，通常与微处理器的字长一致。例如 Intel 8086 微处理器字长 16 位，其数据总线宽度也是 16 位。数据的含义是广义的，它可以是真正运算需要的数据，也可以指令代码或状态信息，有时甚至是一个控制信息，因此，在实际工作中，数据总线上传送的并不一定仅仅是运算数据。

地址总线是专门用来传送地址的，由于地址只能从 CPU 传向外部存储器或 I/O 端口，所以地址总线总是单向三态的。地址总线的位数决定了 CPU 可直接寻址的内存空间大小，比如地址总线为 16 位，则其最大可寻址空间为 $2^{16}=64$KB，地址总线为 20 位，其可寻址空间为 $2^{20}=1$MB。一般来说，若地址总线为 n 位，则可寻址空间为 2^n 个地址空间（存储单元）。例如，我们常用的 32 位处理器多采用 32 位址总线，可以寻址的最大主存空间为 4GB。

控制总线用来传送控制信号和时序信号。控制信号中，有的是微处理器送往存储器和 I/O 接口电路的，如读写信号、片选信号、中断响应信号等；也有其他部件反馈给 CPU 的，如

中断申请信号、复位信号、总线请求信号等。因此，控制总线的传送方向由具体控制信号而定，一般是双向的。控制总线的位数要根据系统的实际控制需要而定，实际上控制总线的具体情况主要取决于CPU。

1.2.2.2 总线的分类

总线按照功能可以分为数据总线、地址总线和控制总线，按照层次、传输方式等不同的角度可以有不同的分类。

总线在各个层次上提供部件之间的连接和信息交换通路，按不同的层次分为3种。

- 内部总线：指芯片内部连接各元件的总线。例如CPU芯片内部，在各个寄存器、ALU、指令部件等各元件之间有总线相连。
- 系统总线：指连接CPU、存储器和各种I/O模块等主要部件的总线，又称为板级总线或板间总线。
- 局部总线：处理器-主存专用总线、高速I/O总线等，这类总线用于主机和I/O设备之间或计算机系统之间的通信。

按总线的数据传输方式可以分为两种。

- 串行总线：在数据线上按位进行传输，只需一根数据线，线路成本低，适合于远距离数据传输。串行总线早期为慢速总线，连接慢速设备，目前已出现了大量高速串行总线，如USB、1394、DP等。
- 并行总线：在数据线上同时有多位一起传送，每一位有一根数据线，故需多根数据线，速度比串行总线快，如ATA、PCI等。

1.2.2.3 串行和并行传输

1）串行传输

当信息以串行方式传输时，只需要一条传输线，且采用脉冲传送。在串行传送时，按顺序传送表示一个数码的所有二进制位（bit）的脉冲信号，每次一位，通常以第一个脉冲信号表示数码的最低有效位，最后一个脉冲信号表示数码的最高有效位。串行传输如图1-5所示。

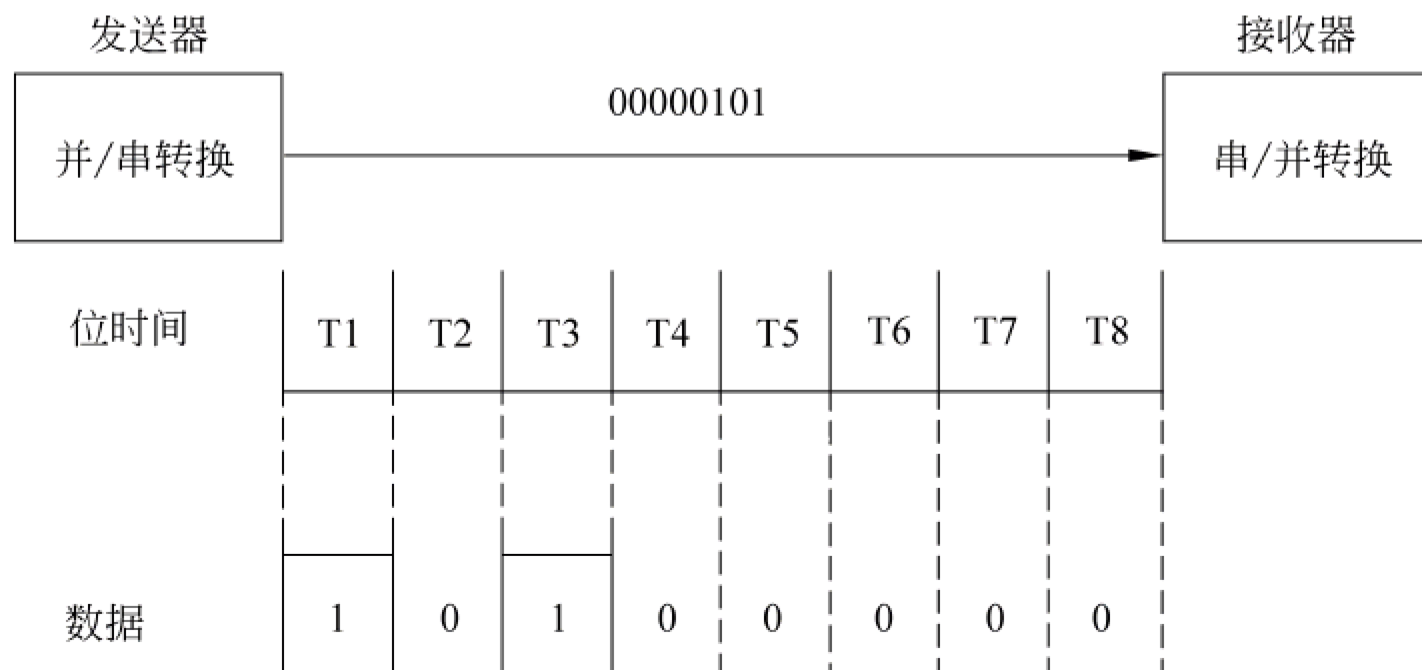


图1-5 串行传输

当串行传输时，有可能按顺序连续传送若干个0或若干个1。为了确定究竟传送了多少个连续的0或连续的1，通常采用的方法是指定"位时间"，即指定一个二进制位在传输线上占用的时间长度，"位时间"通常用一个时钟信号来控制，时钟的频率决定了串行传输的速度。

在串行传输时，被传送的数据需要在发送部件进行并/串变换，而在接收部件又需要进行串/并变换。串行传输的主要优点是只需要一条传输线，这一点对长距离传输尤其重要，不管传送多少数据量都只需要一条传输线，成本比较低廉。

2）并行传输

用并行方式传输二进制信息时，对应于每个数据位都需要一条单独的传输线，信息由多少二进制位组成，就需要多少条传输线，从而使得二进制数0或1在不同的线上同时进行传输，如图1-6所示。

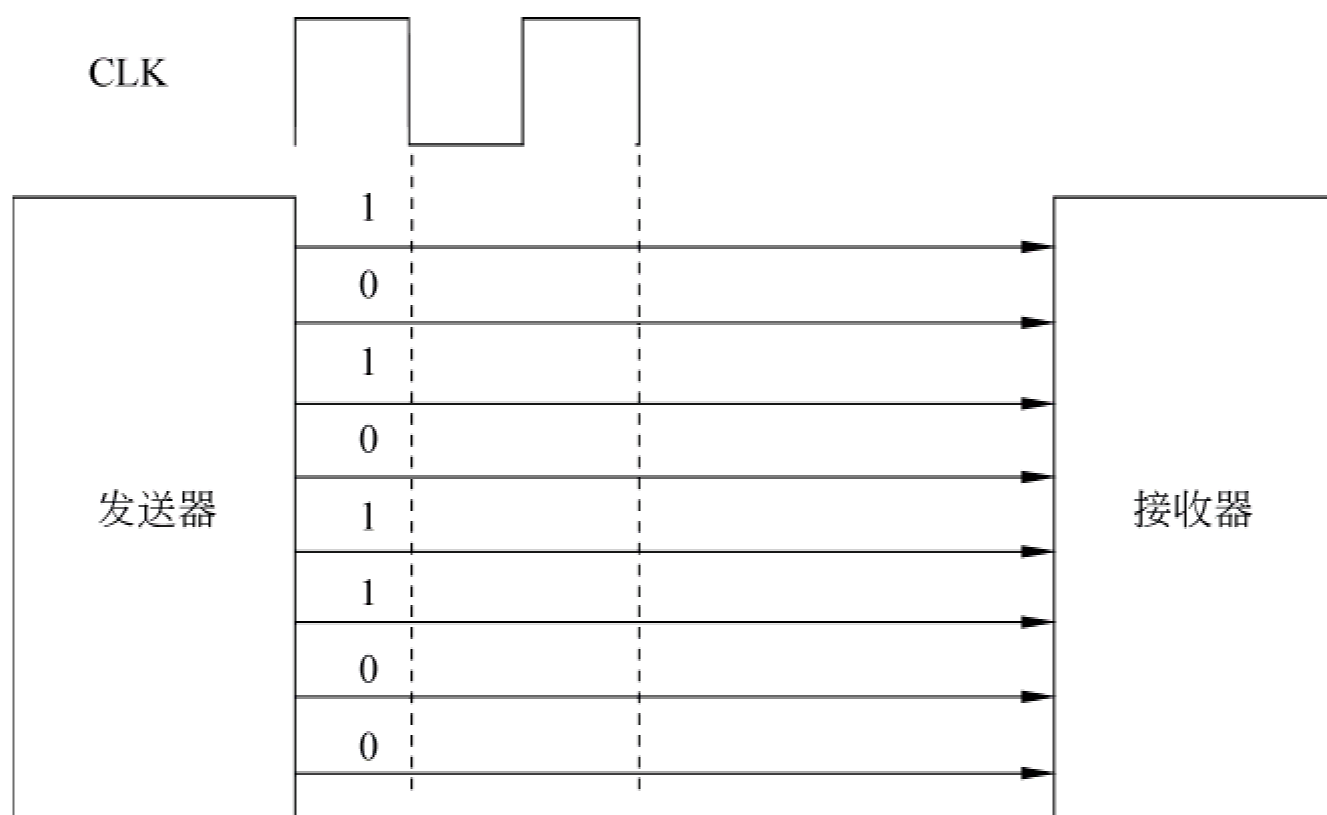


图1-6　并行传输

很明显，并行通信的速度要比串行通信的速度要快，效率更高，费时更少。早期I/O为了提高效率多采用并行总线。但随着总线时钟的提高，在高速状态下，并行口的数据线之间存在串扰，且并行口需要信号同时发送同时接收，任何一根数据线的延迟都会引起问题。而串行只有一根数据线，还可以采用差分信号传输提高抗干扰性，所以可以实现更高的传输速率；因此尽管并行可以一次传多个数据位，但是时钟远远低于串行，所以目前串行传输是CPU和外设高速传输的首选方案。

1.2.2.4　总线的连接方式

总线的排列布置以及总线与其他各类部件的连接方式，对计算机系统性能有重要影响。根据连接方式的不同，微机系统中采用的总线结构可分成三种基本类型：单总线结构、双总线结构和三总线结构。

1）单总线结构

在一些微机系统中，使用一条系统总线来连接CPU、主存和I/O设备，称为单总线结

构,如图 1-7 所示。

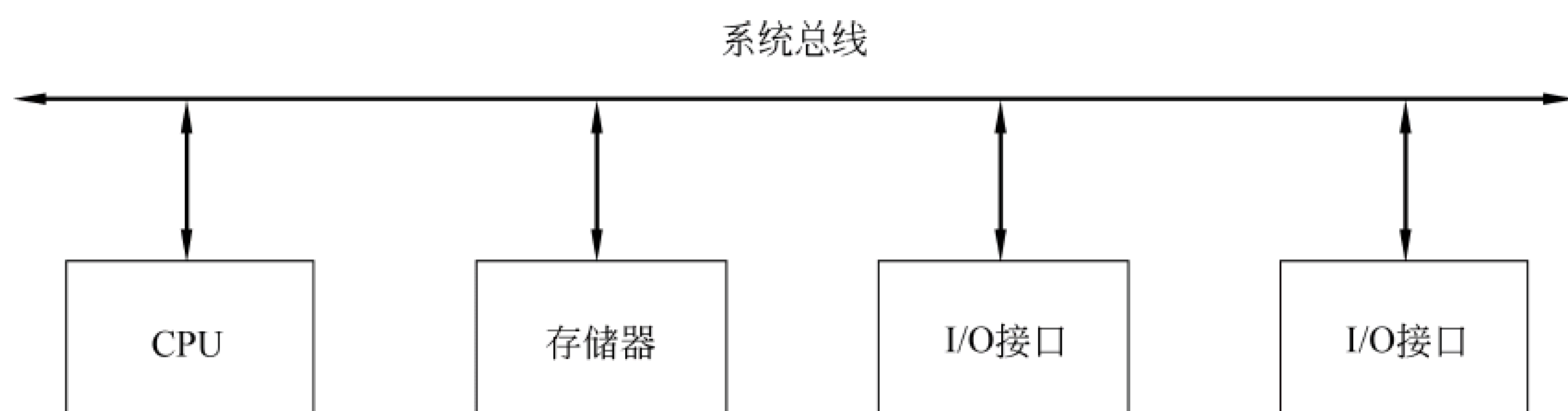


图 1-7　单总线结构

在单总线结构中,要求连接到总线上的逻辑部件都必须高速运行,以便在某些设备需要使用总线时能够迅速获得总线控制权,当不再使用总线时也能迅速放弃总线控制权。否则,由于一条总线由多个功能部件共用,有可能导致很大的时间延迟。

在单总线结构中,当 CPU 取一条指令时,首先把程序计数器(PC)中的地址同控制信息一起送至总线上。该地址不仅送至主存,同时也送至总线上的所有外围设备,只有与总线上的地址相对应的设备才执行数据传送操作。取指令情况下的地址是主存地址,因此该地址所指定的主存单元中的指令被传送给 CPU,CPU 检查指令中的操作码,确定对数据执行什么操作,以及数据是流进还是流出 CPU。

在单总线系统中,对输入输出设备的操作与主存的操作方法完全一样。当 CPU 把指令的地址字段送到总线上时,如果该地址字段对应的地址是外围设备地址,则外围设备予以响应,从而在 CPU 和对应的外围设备之间发生数据传送,数据传送的方向也由指令操作码决定。

单总线结构的优点在于容易扩展成多 CPU 系统,只要在系统总线上挂接多个 CPU 即可。但是,在单总线结构中,由于所有逻辑部件都挂在同一个总线上,因此总线只能分时工作,即某一个时间只能允许一对部件之间传送数据,这就使信息传送的吞吐量受到限制。

2) 双总线结构

图 1-8 为双总线结构,这种结构保持了单总线系统简单、易于扩充的优点,但又在 CPU 和主存之间专门设置了一组高速的存储总线,使 CPU 可通过专用的存储总线与存储器交换信息,以减轻系统总线的负担,同时主存仍可通过系统总线与外设进行 DMA(Direct Memory Access)操作,而不必经过 CPU。

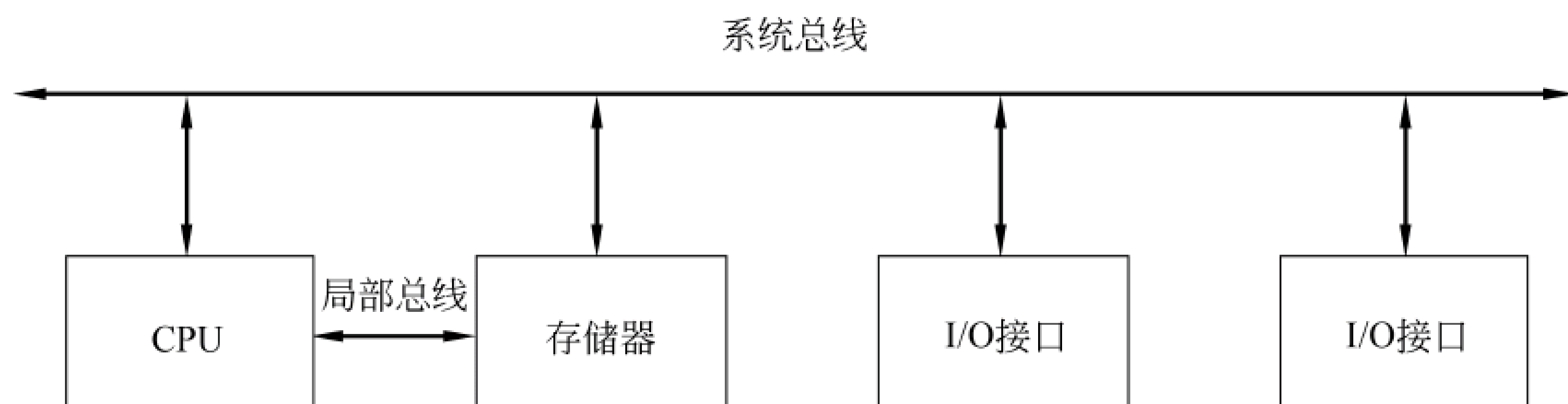


图 1-8　双总线结构

3）三总线结构

图 1-9 为三总线结构。三总线结构是在双总线系统的基础上增加 I/O 总线形成的，其中系统总线是 CPU、主存和 I/O 通道处理机（IOP）之间进行数据传送的公共通路，而 I/O 总线则是多个外围设备与通道之间进行数据传送的公共通路。

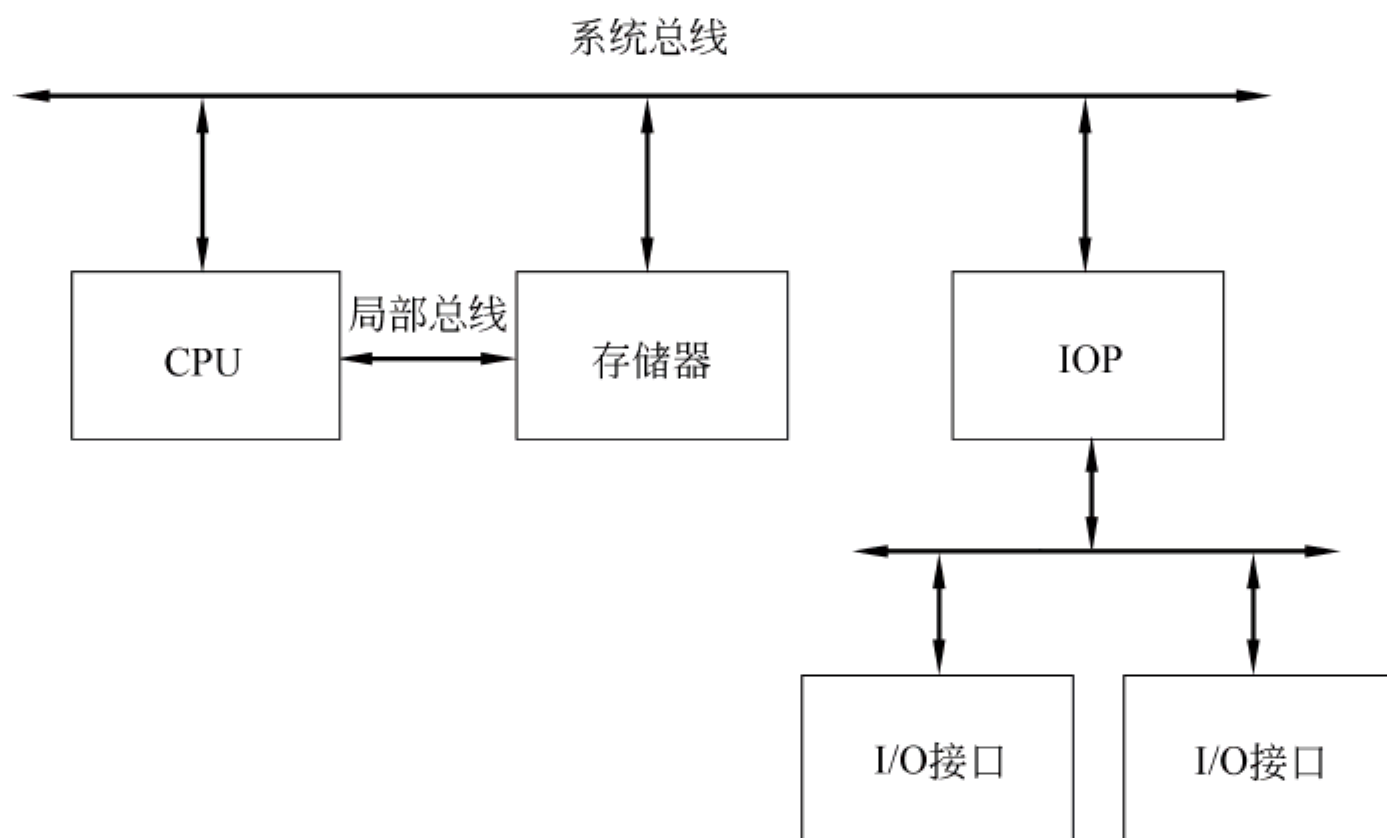


图 1-9　三总线结构

通道实际上是一台具有特殊功能的处理器，又称为 IOP（I/O Processor），它分担了 CPU 的一部分功能，实现对外设的统一管理，完成外设与主存之间的数据传送。这一思想与基于总线的网络将集线器（Hub）转换成交换机（Switch）以提高通信速率的思想是一致的。显然，由于增加了 IOP，整个系统的工作效率可以大大提高。

1.2.2.5　典型总线

1）工业标准结构总线（Industry Standard Architecture，ISA）

ISA 总线是由 IBM PC/XT 和 PC/AT 使用的 8 位总线发展而来的总线标准。ISA 是 8/16 位兼容总线，因此 I/O 插槽有 8 位和 8/16 位两种类型：8 位扩展槽由 62 个引脚组成，其中包括 20 条地址线和 8 条数据线，用于 8 位数据传输；8/16 位扩展槽除了一个 8 位 62 线的连接器外，还有一个附加的 36 线连接器，这种扩展插槽既可以支持 8 位插接板，也可支持 16 位插接板（24 条地址线和 16 条数据线）。ISA 总线的应用范围很广，一般用于连接中、低速 I/O 设备。

2）外部设备连接总线（Peripheral Component Interconnect，PCI）

PCI 总线是 1991 年由 Intel、IBM、Compaq、Apple 等几家公司联合推出的。PCI 是一个与处理器无关的高速外围总线，又是至关重要的层间总线，可支持 10 台外部设备，它采用同步时序协议和集中式仲裁策略，并具有自动配置能力。PCI 总线体系结构中有三种桥接器：HOST 桥、PCI-PCI 桥、PCI-LEGACY 桥。桥接器是一个总线连接和转换部件，可以把一条总线的地址空间映射到另一条总线的地址空间上，从而使系统中任意一个总线主设备都能看到同样的一份地址表。HOST 桥是 PCI 总线控制器和仲裁器。

3) PCI Express 总线

随着 Pentium 4 前端总线频率的迅速提高(高达 1GHz 以上),原有的 PCI 总线标准已难以适应新的要求。PCI Express 是一种基于串行技术、高带宽连接点、芯片到芯片连接的新型总线技术。有别于 PCI 并行技术,PCI Express 的一个通道采用 4 根信号线,两根差分信号线用于接收,另外两根差分信号线用于发送;信号频率 2.5GHz,采用 8/10 位编码;定义了用于多种通道的连接方式,如×1、×4、×8、×16 以及×32 通道的连接器,分别对应于 500MB/s、2GB/s、4GB/s、8GB/s 和 16GB/s 的带宽。采用 PCI Express 总线标准的最大意义在于其通用性和兼容性,通过与 PCI 软件模块的完全兼容,可以确保现有设备和驱动程序不用修改仍能正常工作。

4) 通用串行总线(Universal Serial Bus,USB)

USB 总线是连接计算机系统与外部设备的一种串口总线标准,也是一种输入输出接口的技术规范,被广泛地应用于个人计算机和移动设备等信息通信产品,并扩展至摄影器材、数字电视(机顶盒)、游戏机等其他相关领域。

USB 最初是由英特尔公司与微软公司倡导发起,其最大的特点是支持热插拔和即插即用。当设备插入时,主机枚举到此设备并加载所需的驱动程序,因此在使用上远比 PCI 和 ISA 总线方便。USB 的设计为非对称式的,它由一个主机控制器和若干通过集线器设备以树形连接的设备组成。一个控制器下最多可以有 5 级 Hub,包括 Hub 在内,最多可以连接 128 个设备,一台计算机可以同时有多个控制器。USB 1.1 的最大传输带宽为 12Mb/s,USB 2.0 的最大传输带宽为 480Mb/s,USB 3.0 为 5Gb/s。最新一代是 USB 3.1,传输速度为 10Gb/s,三段式电压 5V/12V/20V,最大供电 100W,另外除了旧有的 Type-A、B 接口之外,新型 USB Type-C 接头不再分正反。

5) SATA 总线

SATA(Serial Advanced Technology Attachment)是一种替代并行 ATA 的总线技术。SATA 总线使用嵌入式时钟信号,具备了更强的纠错能力,与以往相比其最大的区别在于能对传输指令(不仅仅是数据)进行检查,如果发现错误会自动矫正,这在很大程度上提高了数据传输的可靠性。串行接口还具有结构简单、支持热插拔的优点。SATA 分别有 SATA 1.5Gb/s、SATA 3Gb/s 和 SATA 6Gb/s 三种规格,是目前 PC 硬盘连接的主流总线。

1.2.3 哈佛体系结构

冯·诺依曼结构只有一个存储器,数据和指令存储在同一个存储器中,使用同一组地址线进行数据和指令的读写。代表性的处理器有 Intel 8086 及后续系列和 ARM7 等。

哈佛架构(Harvard Architecture)是一种将程序指令和数据分开的存储器结构。程序指令储存和数据储存具有独立的总线接口,数据和指令的访问可以同时进行。

哈佛架构的微处理器通常具有较高的执行效率,程序指令和数据指令分开组织和储存的,可以实现指令预取。目前,大多数处理器采用这种独立信号通路的结构。

如图 1-10 所示,纯冯·诺依曼架构下的 CPU 可以读取指令或读写主存数据,指令和数

据分时共享相同的总线。使用哈佛结构的 CPU，即使没有缓存的情况下，也可以读取指令的同时进行数据访问，因此，哈佛结构的计算机可以在相同的电路复杂度下有更好的性能表现，哈佛架构拥有不同的代码和数据的地址空间：即指令的零地址和数据的零地址是不同的。

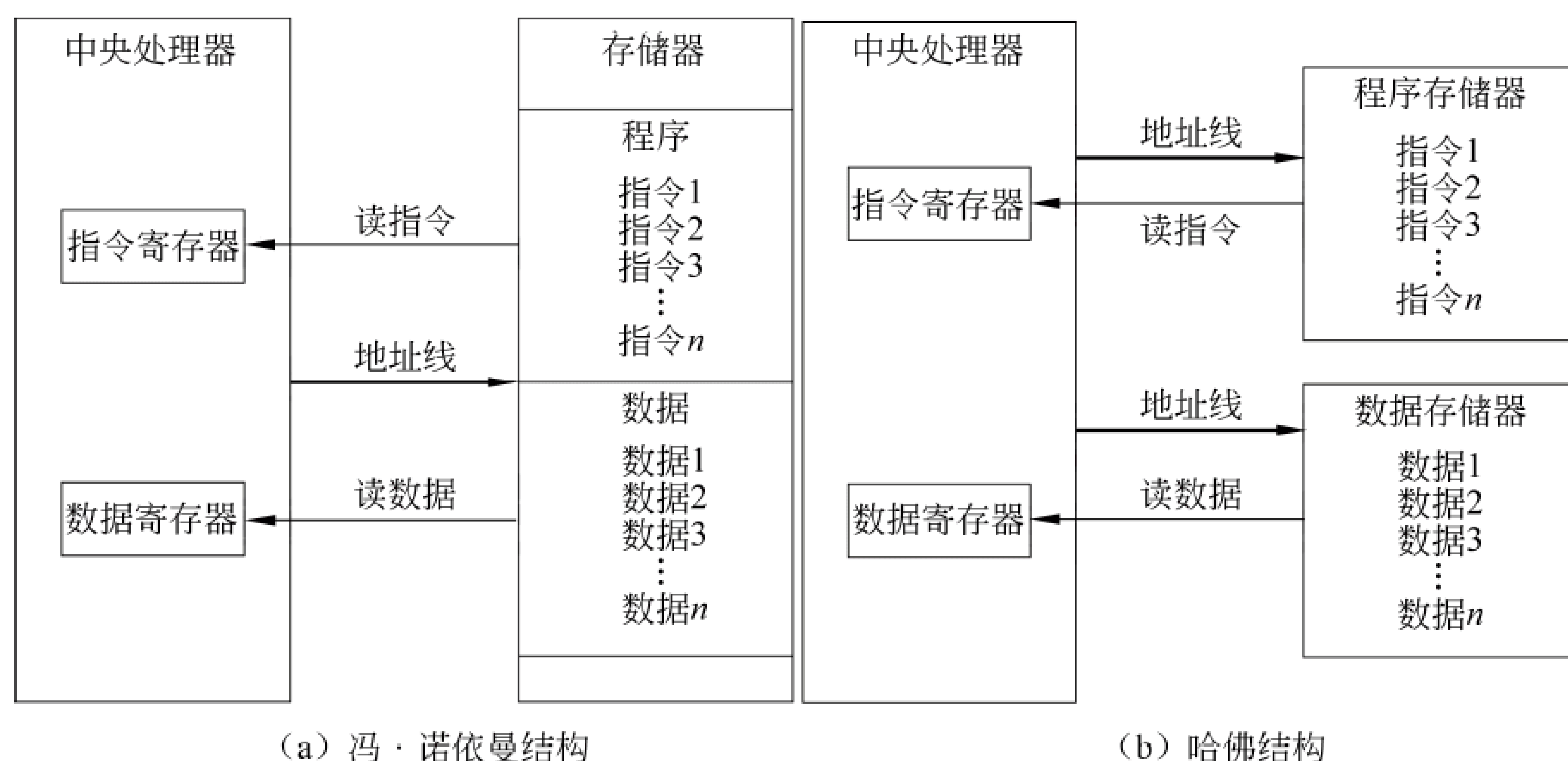


图 1-10　冯 · 诺依曼结构与哈佛结构的区别

现代高性能 CPU 芯片在设计上融合了哈佛结构和冯 · 诺依曼结构的特点，一种典型的方式是将 CPU 的缓存分为指令缓存和数据缓存两部分，CPU 访问缓存时使用哈佛体系结构；然而当缓存未命中时，数据从主存储器中读取，此时并不分为独立的指令和数据部分。

哈佛结构主要用于数字信号处理器（DSP）和微控制器。DSP 一般执行高度优化的音频或视频处理算法，通常采用单指令多数据流（SIMD）和超长指令字（VLIW）等架构，一般具有多套总线实现数据的并行读写，例如 TI 的 TMS320 C55x 处理器，具有多个并行数据总线（双写、三读）和指令总线。

微控制器的特点是具有少量的程序存储器（闪存）和数据存储器（SRAM），较少有缓存，大多利用哈佛架构的并行高速处理指令和数据的访问。分开存储的程序和数据存储器可能具有不同的位宽，例如使用 16 位指令和 8 位宽的数据，如 Atmel 的 AVR 和 Microchip 的 PIC 系列。

1.2.4　微处理器的内部结构

1.2.4.1　基本结构

传统上，CPU 由控制器和运算器这两个主要部件组成。随着集成电路技术的不断发展和进步，新型 CPU 纷纷集成了一些原先置于 CPU 之外的分立功能部件，如浮点处理器、高速缓存等，在大大提高 CPU 性能指标的同时，也使得 CPU 的内部组成日益复杂化。

1. 控制器

控制器是整个计算机系统的指挥中心。在控制器的指挥控制下，运算器、存储器和输入

输出设备等部件协同工作。

控制器根据程序预定的指令执行顺序，从主存取出一条指令，按照该指令的功能，控制计算机内各功能部件的操作，协调和指挥整个计算机实现指令的功能。

控制器通常由程序计数器(PC)、指令寄存器(IR)、指令译码器(ID)和操作控制器组成。其主要功能包括：

(1) 从主存中取出一条指令，并指出下一条指令在主存中的位置；

(2) 对指令进行译码，并产生相应的操作控制信号，以便启动规定的动作；

(3) 指挥并控制 CPU、主存和输入输出设备之间数据流动的方向。

2. 运算器

运算器是计算机中用于实现数据加工处理功能的部件，它接受控制器的命令，负责完成对操作数据的加工处理任务，其核心部件是算术逻辑单元 ALU。运算器接受控制器的命令而进行动作，即运算器所进行的全部操作都是由控制器发出的控制信号来指挥的，所以它是执行部件。

运算器由算术逻辑单元(ALU)、寄存器、程序状态寄存器等组成。它有两个主要功能：

(1) 执行所有的算术运算；

(2) 执行所有的逻辑运算，并进行逻辑测试。

3. 寄存器

在 CPU 中至少要有五类寄存器：指令寄存器(IR)、程序计数器(PC)、地址寄存器(AR)、数据寄存器(DR)、程序状态寄存器(PSR)。这些寄存器用来暂存一个计算机的字，其数目可以根据需要进行扩充。

1) 数据寄存器(Data Register，DR)

数据寄存器又称数据缓冲寄存器，其主要功能是作为 CPU 和主存、外设之间信息传输的中转站，用于弥补 CPU 和主存、外设之间操作速度上的差异。

数据寄存器用来暂时存放由主存储器读出的一条指令或一个数据字；反之，当向主存存入一条指令或一个数据字时，也将它们暂时存放在数据寄存器中。

数据寄存器的作用：

• 作为 CPU 和主存、外围设备之间信息传送的中转站；
• 弥补 CPU 和主存、外围设备之间在操作速度上的差异。

2) 指令寄存器(Instruction Register，IR)

指令寄存器用来保存当前正在执行的一条指令。当执行一条指令时，首先把该指令从主存读取到数据寄存器中，然后再传送至指令寄存器。

指令包括操作码和操作数/地址码两个字段，为了执行指令，必须对操作码进行测试，识别出所要求的操作，指令译码器(Instruction Decoder，ID)用于完成这项工作。指令译码器对指令寄存器的操作码部分进行译码，以产生指令所要求操作的控制电位，在时序部件定时信号的作用下，产生具体的操作控制信号。

3) 程序计数器(Program Counter，PC)

程序计数器用来指出下一条指令在主存储器中的地址。在程序执行之前，首先必须将

程序的首地址，即程序第一条指令所在主存单元的地址送入 PC。当执行指令时，CPU 能自动递增 PC 的内容，使其始终保存将要执行的下一条指令的主存地址，为取下一条指令做好准备。若为单字节指令，则 PC+1；若为双字节指令，则 PC+2，以此类推。

但是，当遇到转移指令时，下一条指令的地址将由转移指令的地址码字段来指定，即 PC 寄存器的值由指令所带的地址决定而非通过顺序递增 PC 的内容来取得。

4）地址寄存器（Address Register，AR）

地址寄存器用来保存 CPU 当前所访问的主存单元的地址。由于在主存和 CPU 之间存在操作速度上的差异，所以必须使用地址寄存器来暂时保存主存的地址信息，直到主存的存取操作完成为止。当 CPU 和主存进行信息交换，即 CPU 向主存存入数据/指令或者从主存读出数据/指令时，都要使用地址寄存器和数据寄存器。如果我们把外围设备与主存单元进行统一编址，那么，当 CPU 和外围设备交换信息时，我们同样要使用地址寄存器和数据寄存器。

5）程序状态寄存器（Program Status Register，PSR）

程序状态寄存器用来保存由算术/逻辑指令运行或测试的结果所建立起来的各种条件码内容，如运算结果进/借位标志（C）、运算结果溢出标志（O）、运算结果为零标志（Z）、运算结果为负标志（N）、运算结果符号标志（S）等。除此之外，程序状态寄存器还用来保存中断和系统工作状态等信息，以便 CPU 和系统及时了解机器运行状态和程序运行状态。

1.2.4.2　流水线

早期的计算机采用的是串行处理，计算机的各个操作只能串行地完成，即任一时刻只能进行一个操作。并行处理使得多个操作能同时进行，从而大大提高了计算机的速度。并行处理的主要方法包括如下。

（1）时间并行技术：时间并行指分时，即让多个处理过程在时间上相互错开，轮流重叠地使用同一套硬件设备的各个部分，以加快硬件周转而赢得速度。分时并行的实现方式就是流水线，目前的高性能计算机几乎无一例外地使用了流水线技术。

（2）空间并行技术：空间并行指硬件资源重复，即以多个相同的硬件来大幅度提高计算机的处理速度。空间并行技术主要体现在多核处理器、多处理器系统，超算就是典型的空间并行技术。

（3）时间空间并行技术：即综合使用了分时和硬件并行，高性能微处理器一般都采用时间空间并行技术。

计算机的流水线（Pipeline）工作方式就是将一个计算任务细分成若干子任务，每个子任务都由专门的功能部件进行处理，一个计算任务的各个子任务由流水线上各个功能部件轮流进行处理，最终完成工作。这样，不必等到上一个计算任务完成，就可以开始下一个计算任务的执行。

流水线的硬件基本结构如图 1-11 所示。流水线由一系列串联的功能部件（Si）组成，各个功能部件之间设有高速缓冲寄存器（L），以暂时保存上一功能部件对子任务处理的结果，同时又能够接受新的处理任务。在一个统一的时钟（C）控制下，计算任务从功能部件的一

个功能段流向下一个功能段。在流水线中,所有功能段同时对不同的数据进行不同的处理,各个处理步骤并行地操作。

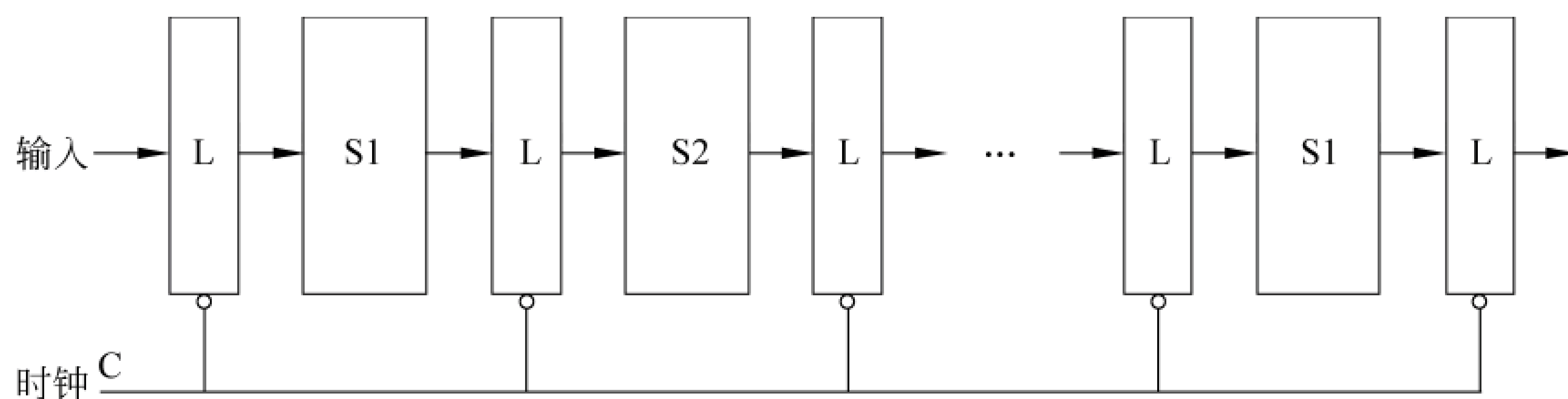


图 1-11　流水线的硬件基本结构

理想情况下,每步需要一个时钟周期。当流水线完全装满时,每个时钟周期平均有一条指令从流水线上执行完毕,输出结果。当指令流不能按流水顺序执行时,流水过程会中断(即断流)。在一个流水过程中,流水线的性能取决于流水部件中最慢的部件,因此实现各个子过程的各个功能段所需要的时间应该尽可能保持相等,以避免产生瓶颈。

假设一个指令的执行周期包含取指(IF)、译码(ID)、执行(EX)、访存(MEM)、写回(WB)5 个过程,最慢的部件执行时间记为一个单位时间,在不采用流水线时的执行过程如图 1-12 所示。上一条指令的 5 个子过程全部执行完毕后才能开始下一条指令,每隔 5 个单位时间才有一个输出结果,15 个单位时间完成 3 条指令,每条指令平均用时 5 个单位时间。

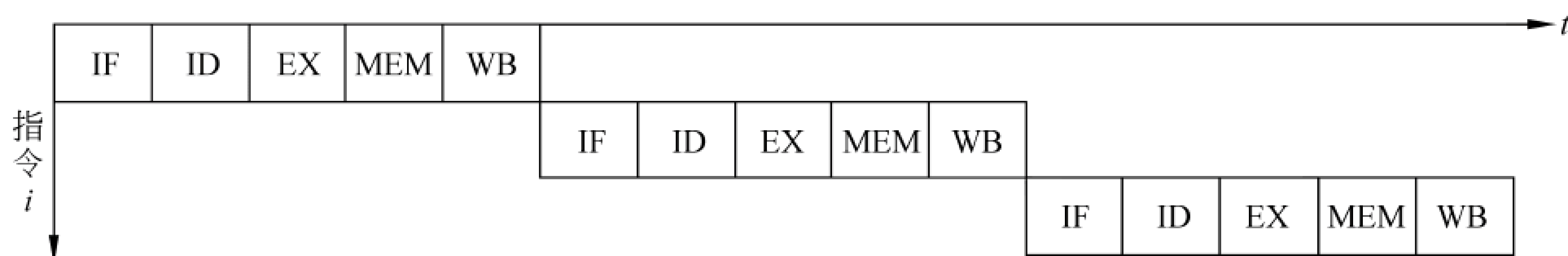


图 1-12　非流水顺序执行过程

采用 5 级流水线后指令的执行过程如图 1-13 所示,上一条指令与下一条指令的 5 个子过程在时间上可以重叠执行,当流水线满载时,每一个单位时间就可以输出一个结果。因此,9 个单位时间完成了 5 条指令,每条指令平均用时 1.8 个单位时间。虽然每条指令的执行时间并未缩短,但 CPU 运行指令的总体速度却能成倍提高。

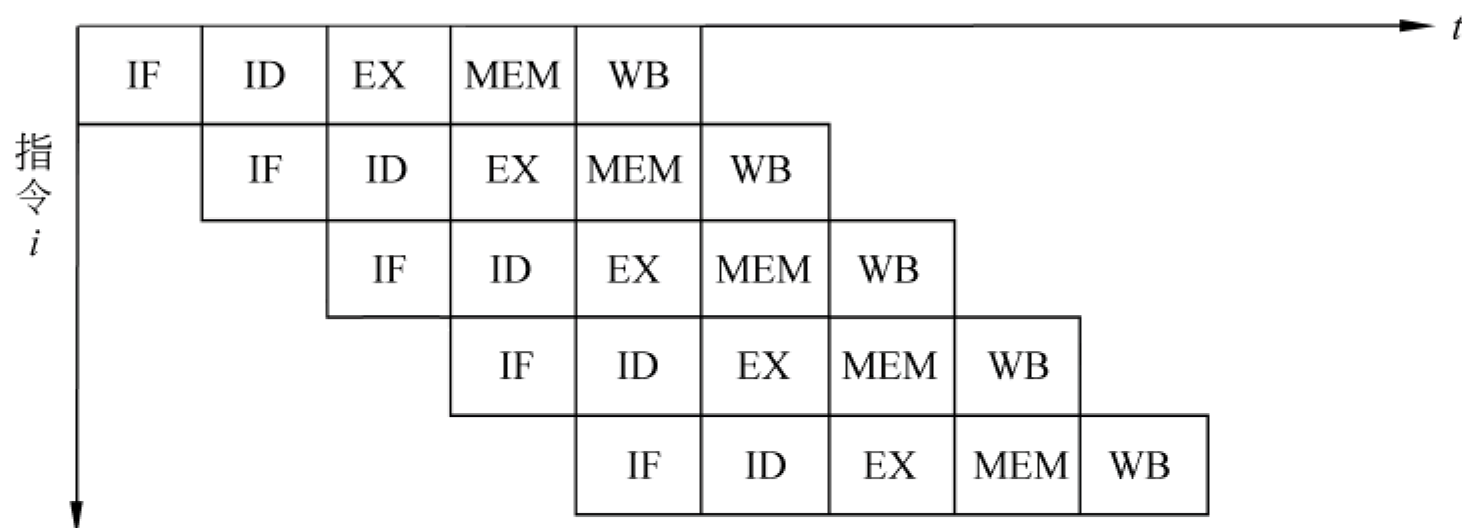


图 1-13　5 级流水执行过程

流水线的每个阶段的计算结果在周期结束以前都要发送到阶段之间的缓冲器上，以供下一个阶段使用。所以，单位时间就是由以上几个阶段中的耗时最长的那个决定的。假设耗时最长的阶段耗时为 s 秒，那么时钟频率最多就只能设计到 1s/Hz。要提高时钟频率，一种最简单的办法，就是将每个阶段再进行细分成更小的步骤，细分后的每个阶段，单个阶段的运算量小了，单位耗时 s 也就减少，这样实际上就是提高了时钟频率。这种将标准流水线细分的技术，就是超级流水线技术。比如，Pentium M 有 14 级流水线，Pentium 4 有 31 级流水线。

只有一条指令流水线的计算机称为标量流水计算机，如果计算机具有两条以上的流水线，称为超标量计算机，图 1-14 表示超标量流水计算机的时空图。当流水线满载时，每一个时钟周期可以执行 2 条以上的指令。超标量流水计算机是时间并行技术和空间并行技术的综合应用。

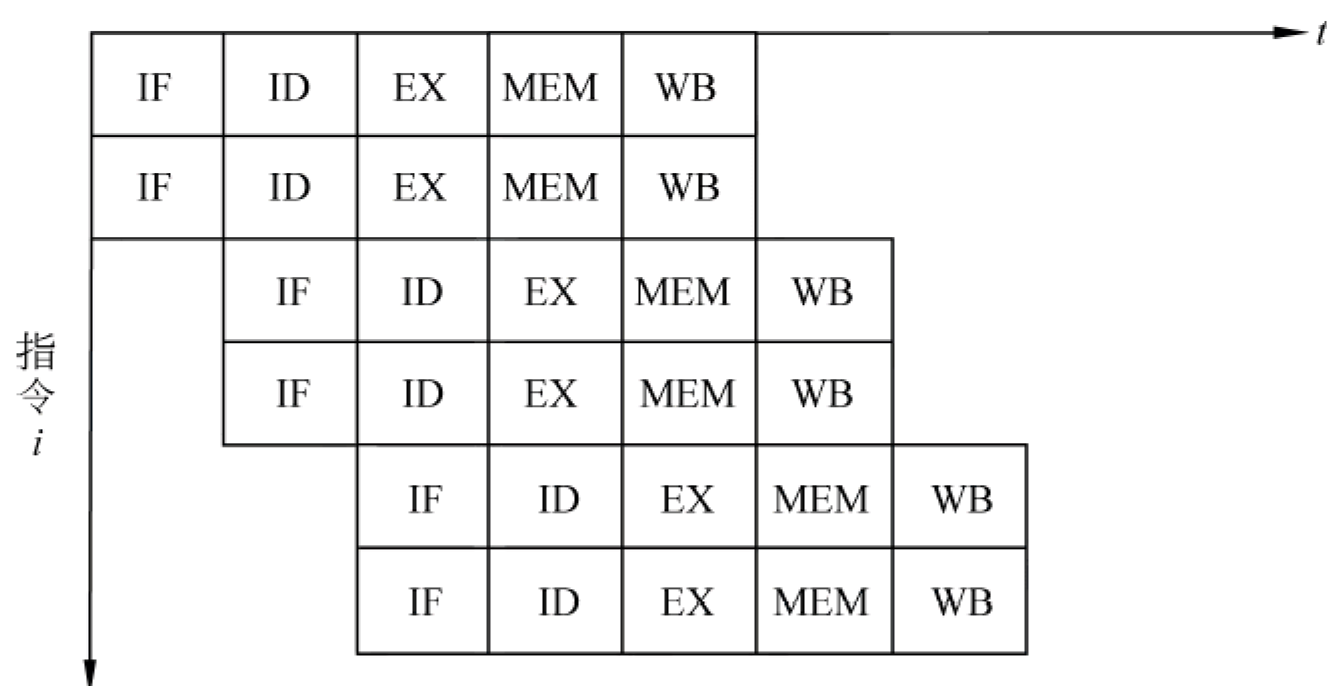


图 1-14　超标量流水线执行

要使流水线发挥高效率，就要使流水线连续不断地流动，尽量不出现断流。然而，流水线中的各条指令之间存在一些相关性，即第二条指令的执行必须要使用第一条指令的执行结果，使得指令的执行受到影响，断流不可避免；此外，跳转指令也会引起流水线断流，现代计算机通常采用乱序执行和分支预测等进行执行优化，以降低断流引起的性能损失。

【**思考题：**硬件多线程是什么？属于空间并行还是时间并行？】

1.2.5　I/O 接口技术

一般而言，CPU 管理外围设备的输入输出控制方式有 5 种：查询方式、中断方式、DMA 方式、通道方式、外围处理机方式。第一种方式由软件实现，后四种方式需要特殊硬件的支持才能实现。目前微机系统中常用的是查询方式、中断方式和 DMA 方式。

1）查询方式

程序查询方式是早期计算机中使用的一种方式，CPU 与外围设备的数据交换完全依赖于计算机的程序控制。在进行信息交换之前，CPU 要设置传输参数、传输长度等，然后启动外设工作，与此同时，外设则进行数据传输的准备工作；相对于 CPU 来说，外设的速度是比

较低的,因此外设准备数据的时间往往是一个漫长的过程,而 CPU 不知道数据何时准备好,在这段时间里,CPU 除了循环检测外设是否已准备好之外,不能处理其他业务,只能一直等待,直到外设完成数据准备工作,CPU 才能开始进行信息交换。

查询方式的优点是 CPU 的操作和外围设备的操作能够完全同步,不需要额外硬件。但是,由于外围设备的动作通常很慢,程序进行循环查询浪费 CPU 时间,数据传输效率低下,CPU 利用率很低。在当前的实际应用中,除了简单的嵌入式系统应用之外,已经很少使用程序查询方式了。

【思考题:从用户和操作系统的角度分别考虑,CPU 利用率高好还是低好?】

2) 中断方式

中断是外围设备用来主动通知 CPU,准备发送或接收数据的一种方式。当一个中断发生时,CPU 暂停其现行程序,转而执行中断处理程序,完成数据 I/O 工作;当中断处理完毕后,CPU 又返回到原来的任务,并从暂停处继续执行程序。这种方式节省了 CPU 时间,实现了外设和 CPU 的并行工作,是管理 I/O 操作的一个比较有效的方法,现代计算机中,中断是必不可少的 I/O 机制,通常有中断控制器对中断进行管理。中断方式一般适用于随机出现的服务请求,并且对响应速度有一定的要求。

3) 直接存储器存取方式 DMA

直接存储器存取方式(DMA)是一种完全由硬件执行 I/O 交换的工作方式。DMA 控制器从 CPU 完全接管对总线的控制权,数据交换不经过 CPU 而直接在主存和外围设备之间进行,以便高速传送数据。这种方式的主要优点是数据传送效率很高,传送速率仅受限于主存的访问时间,CPU 只需要配置 DMA 数据传输的起始地址,目的地址和数量,数据的传输完全由 DMA 控制总线完成,并通过中断方式通知 CPU。与程序中断方式相比,这种方式需要特殊的硬件支持,适用于主存和高速外围设备之间大批量数据交换的场合。

4) 通道方式

DMA 方式的出现减轻了 CPU 对 I/O 操作的控制,使得 CPU 的效率显著提高,而通道的出现则进一步提高了 CPU 的效率。通道分担了 CPU 的一部分功能,可以实现对外围设备的统一管理,完成外围设备与主存之间的数据传送。

5) 外围处理机方式

外围处理机(Peripheral Processor Unit,PPU)方式是通道方式的进一步发展。PPU 基本上独立于主机工作,它的结构更接近于一般的处理机,甚至就是微小型计算机。在一些系统中,设置了多台 PPU,分别承担 I/O 控制、通信、维护诊断等任务,从某种意义上说,这种系统已经变成了分布式多机系统。

1.2.6 存储器

冯·诺依曼计算机以存储器为中心,必须先把有关程序和数据装到存储器中,程序才能开始运行。在程序执行过程中,CPU 所需的指令、运算器所需的数据要从存储器中取出,运算结果必须在程序执行完毕之前全部写到存储器中,各种输入输出设备也直接与存储器交

换数据。因此，在计算机运行过程中，存储器是各种信息存储和交换的中心。

1. 存储器指标

存储器用于存储二进制数据，其最小的存储单位是一比特，基于地址总线宽度和存储代价的考虑，计算机系统中以8个二进制位即一字节(Byte)作为存储的基本单元，存储器的容量以字节为单位。对于大容量的存储，常用KB、MB、GB、TB来表示。

【**思考题**：以Byte作为基本存储单位合理吗?】

CPU向存储器读写数据时，通常以字(Word)为单位，字由若干个字节组成，一个字到底等于多少个字节取决于计算机的字长，对于32位机，1 Word＝4Byte＝32bit。

衡量存储器的指标主要包括以下几点。

1) 存储容量

在一个存储器中可以容纳的存储单元的总数称为存储容量。在按字节寻址的计算机中，存储容量的最大字节数可由地址码的位数来确定。例如，一台计算机的地址码为n位，则可产生2^n个不同的地址码，则其最大容量为2^n字节。一台计算机设计定型以后，其地址总线、地址译码范围也已确定，因此其最大存储容量是确定的，通常情况下主存储器的实际存储容量小于理论上的最大容量。

【**思考题**：对于辅存，如硬盘，存储容量和地址线有何关系?】

2) 存取时间

存取时间又称为存储器访问时间或读写时间，是指从启动一次存储器操作到完成该操作所经历的时间。从一次读操作命令发出到该操作完成，将数据读入数据缓冲寄存器为止所经历的时间，为存储器读取时间。存储器从接受写命令到把数据从存储器数据寄存器的输出端传送到存储单元所需的时间，即为存储器的写入时间。

3) 存储周期

存储周期又称为访问周期，是指连续启动两次独立的存储器操作所需间隔的最小时间，它是衡量主存储器工作性能的重要指标。存储周期通常略大于存取时间。

【**思考题**：参考DRAM的特点，为何存储周期通常大于存取时间?】

4) 存储器带宽

存储器带宽是指单位时间里存储器所存取的信息量，是衡量数据传输速率的重要指标，通常以位/秒(b/s)或字节/秒(B/s)为单位，与存储器接口的数据总线宽度和存储周期有关。例如，总线宽度为32位，存储周期为250ns，则存储器带宽＝32b/250ns＝128Mb/s。

2. 存储器分类

根据存储介质，存储器分为半导体存储器、磁盘存储器和光盘存储器等。按照存取方式可分为随机存储和顺序存储。按存储器的读写功能可分为只读存储器(Read Only Memory，ROM)和随机读写存储器(Random Access Memory，RAM)。

1) 随机读写存储器

存储单元的内容可按需随意取出或存入，且存取的速度与存储单元的位置无关的存储器。这种存储器在断电时将丢失其存储内容，故主要用于存储短时间使用的程序。按保存数据的机理，随机存储器又分为静态随机存储器(Static RAM，SRAM)和动态随机存储器

(Dynamic RAM,DRAM)。

静态随机存储器(SRAM):只要不断电信息就不会丢失。静态存储器的集成度低,成本高,功耗较大,通常作为 Cache 的存储体。

动态随机存储器(DRAM):采用 MOS 管和电容存储电荷来保存信息,使用时需要不断给电容充电才能保持信息。动态存储器电路简单,集成度高,成本低,功耗小,但需要反复进行刷新操作,工作速度较慢,适合作为主存储器的主体部分。

2) 只读存储器

只读存储器 ROM 是一种存储固定信息的存储器,其特点是在正常工作状态下只能读取数据,不能随时修改或重新写入数据。只读存储器电路结构简单,且存放的数据在断电后不会丢失,特别适合于存储永久性的、不变的程序代码或数据。只读存储器包括不可重写只读存储器和可重写只读存储器(PROM)两大类。我们目前在微机系统里常用的可重写只读存储器主要包括电擦除可编程 ROM(EEPROM)和闪存(Flash ROM)两种。

Flash ROM 中的内容或数据不像 RAM 一样需要电源支持才能保存,但又像 RAM 一样具有可重写性:在某种低电压下,其内部信息可读不可写,类似于 ROM,而在较高的电压下,其内部信息可以更改和删除,类似于 RAM。由于单片 Flash 存储容量大,易于修改,Flash ROM 也常用于数码相机、U 盘以及固态硬盘中。

按存储器在计算机系统中的作用分,存储器可分为主存储器和辅助存储器。CPU 能直接访问的存储器称为内存储器(简称内存),或主存储器。CPU 不能直接访问的存储器称为外存储器(简称外存)或辅助存储器,外存的信息必须调入内存才能被 CPU 使用。

高速缓冲存储器(Cache)是计算机系统中的一个高速、小容量的半导体存储器,通常采用 SRAM,它位于高速的 CPU 和低速的主存之间,用于匹配两者的速度,达到高速存取指令和数据的目的。和主存相比,Cache 的存取速度快,但存储容量小。

内存储器用来存放计算机正在执行的大量程序和数据。

外存储器用于存放系统中的程序、数据文件及数据库。与内存相比,外存的特点是存储容量大,位成本低,但访问速度慢。

3. 分级存储体系

计算机对存储器的要求是容量大、速度快、成本低,需要尽可能地同时兼顾这三方面的要求。但是一般来讲,存储器速度越快,价格也越高,因而也越难满足大容量的要求。目前通常采用多级存储器体系结构,使用高速缓冲存储器、主存储器和外存储器,如图 1-15 所示。

由 Cache 和主存储器构成的 Cache-主存系统,其主要目标是利用与 CPU 速度接近的 Cache 来高速存取指令和数据以提高存储器的整体速度,从 CPU 角度看,Cache-主存的存储体系的访存速度接近 Cache,而容量是主存的容量。由主存和外存构成的虚拟存储器系统,其主要目的是增加存储系统的容量,从整体上看,其速度接近于主存的速度,其容量则接近于外存的容量。计算机存储系统的这种多层次结构,很好地解决了容量、速度、成本三者之间的矛盾。这些不同速度、不同容量、不同价格的存储器,用硬件、软件或软硬件结合的方式连接起来,形成一个系统。这个存储系统对应用程序员而言是透明的,在应用程序员看来

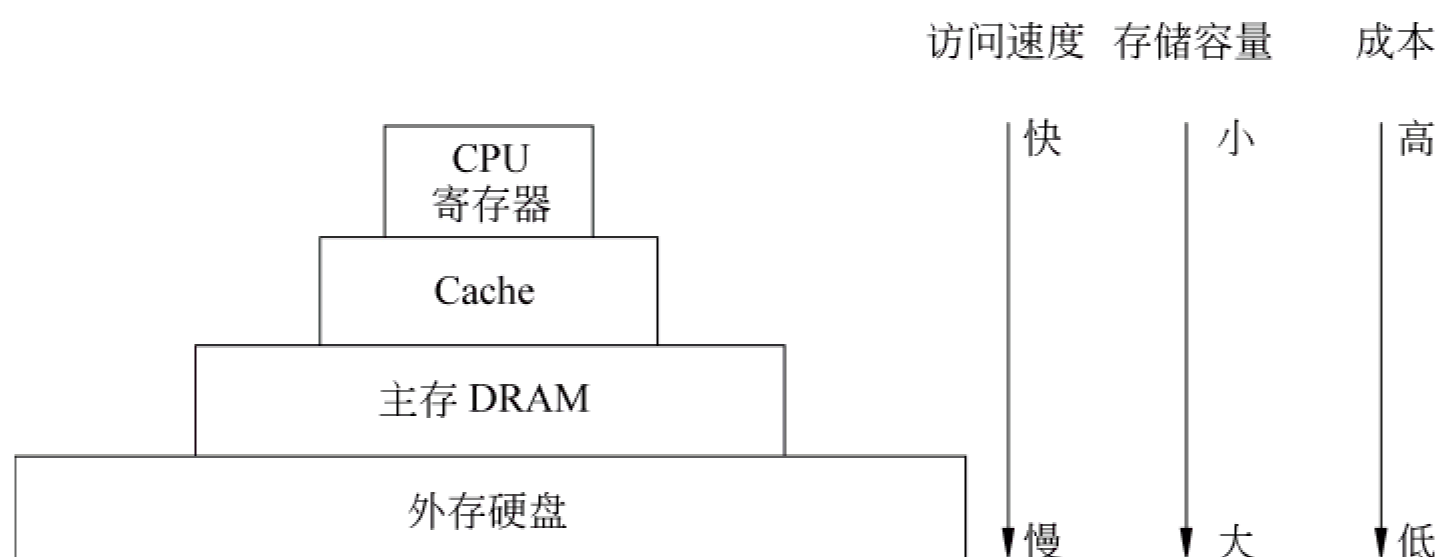


图 1-15　分层存储体系

它是一个存储器，其速度接近于最快的那个存储器，存储容量接近于容量最大的那个存储器，单位价格则接近最便宜的那个存储器。

1）Cache 缓存技术

设计和开发系统程序和应用程序时，程序员通常采用模块化的程序设计方法。某一模块的程序，往往集中在存储器逻辑地址空间中很小的一块范围内，且程序地址分布是连续的。也就是说，CPU 在一段较短的时间内，是对连续地址的一段很小的主存空间频繁地进行访问，而对此范围以外地址的访问较少，这种现象称为程序访问的局部性。

Cache 技术就是利用程序访问的局部性原理，把程序中正在使用的部分（活跃块）存放在一个小容量的高速 Cache 中，使 CPU 的访存操作大多针对 Cache 进行，从而解决高速 CPU 和低速主存之间速度不匹配的问题，使程序的执行速度大大提高。

Cache 是主存的缓冲存储器，由高速的 SRAM 组成，所有控制逻辑全部由硬件实现，对程序员而言是透明的。随着半导体器件集成度的不断提高，当前有些 CPU 已内置 Cache，并且出现了两级以上的多级 Cache 系统。

CPU 与 Cache 之间的数据交换是以字为单位的，而 Cache 与主存之间的数据交换则是以块为单位的。一个块由若干字组成。当 CPU 读取主存中的一个字时，该字的主存地址被发给 Cache 和主存，此时，Cache 控制逻辑依据地址判断该字当前是否存在于 Cache 中：若在，该字立即被从 Cache 传送给 CPU；若不在，则用主存读周期把该字从主存读出送到 CPU，同时把含有这个字的整个数据块从主存读出送到 Cache 中，并采用一定的替换策略将 Cache 中的某一块替换掉，替换算法由 Cache 管理逻辑电路来实现。

2）虚存技术

由于工艺和成本的原因，主存的容量受到限制。然而，计算机系统软件和应用软件的功能不断增强，程序规模迅速扩大，要求主存的容量越大越好，为此，操作系统将部分外存作为主存使用，即一个程序的部分地址空间在主存，另一部分在外存，当所访问的信息不在主存时，则由操作系统来安排 I/O 指令，把信息从外存调入主存。从效果上来看，好像为用户提供了一个存储容量比实际主存大得多的存储器，用户无需考虑所编程序在主存中是否放得下或放在什么位置等问题，这种存储器称为虚拟存储器。虚拟存储器只是一个容量非常大的存储器的逻辑模型，不是任何实际的物理存储器。虚拟存储技术中程序指令采用虚拟地址（或者叫逻辑地址），程序运行时，CPU 以虚拟地址来访问主存，由

辅助硬件找出虚拟地址和实际物理地址之间的对应关系，并判断这个虚拟地址指示的存储单元内容是否已装入主存。如果已在主存中，则通过地址变换，CPU 可直接访问主存的实际单元；如果不在主存中，则把包含这个字的一个外存存储块调入主存后再由 CPU 访问。如果主存已满，则由替换算法从主存中将暂不运行的一块调回外存，再从外存调入新的一块到主存。

1.2.7 程序的执行过程

CPU 的基本工作是执行预先存储的指令序列。程序的执行过程实际上是不断地取出指令、分析指令、执行指令的过程。

CPU 从存放程序的主存储器里取出一条指令，译码并执行这条指令，保存执行结果，紧接着又去取指令，译码，执行指令……如此周而复始，反复循环，使得计算机能够自动地工作，直到遇到停止指令，其过程如图 1-16 所示。

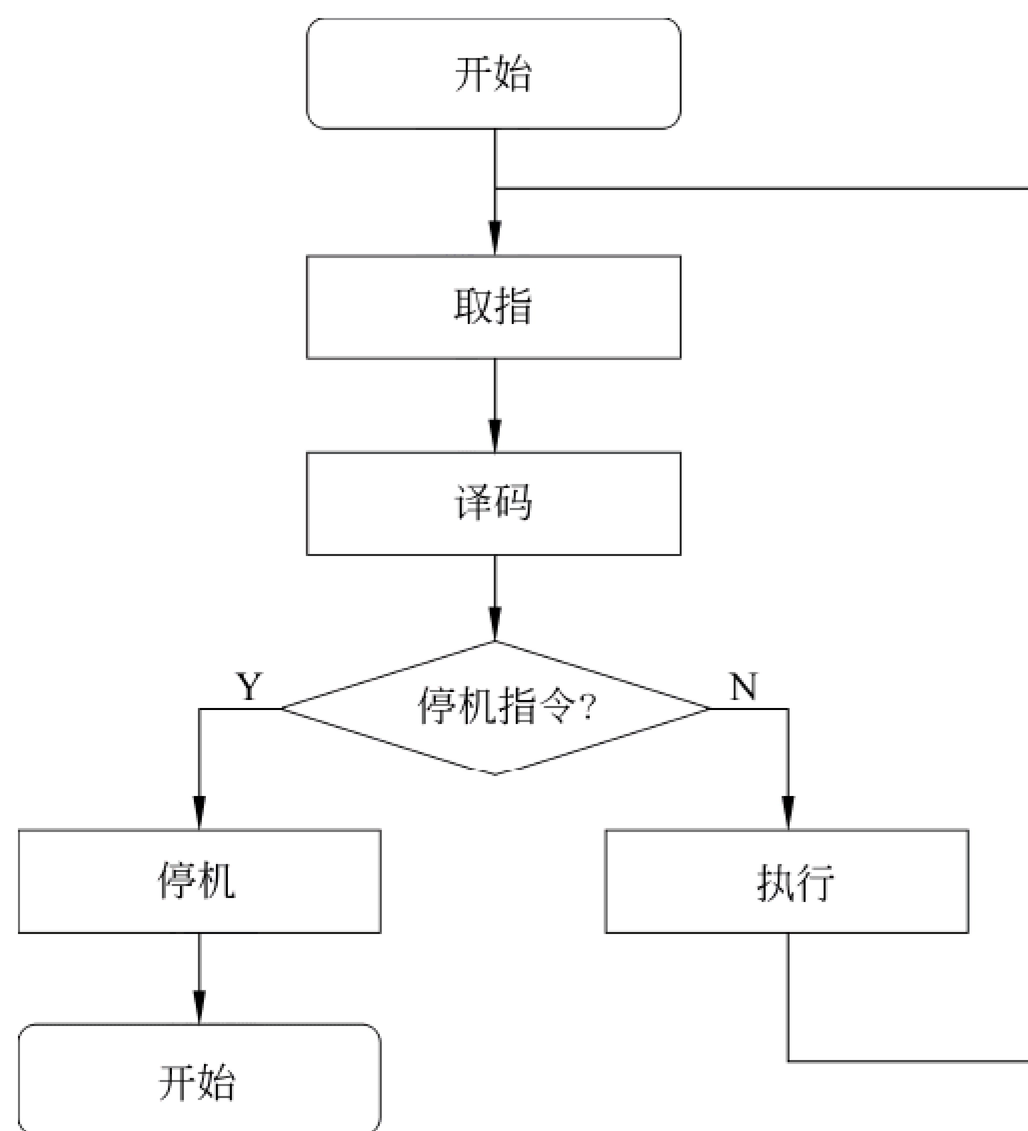


图 1-16 程序执行流程

CPU 是在时钟的驱动下工作的，时钟周期是处理操作的最基本时间单位，由机器的主频决定。机器内部各种操作大致可归属为 CPU 内部操作和对主存的操作两大类。从内存读取一条指令字的最短时间定义为 CPU 周期（也叫机器周期），由于 CPU 内部操作速度较快，CPU 访问一次内存所花的时间较长，CPU 周期包含若干个时钟周期。CPU 取出一条指令并执行该指令所需的时间，称为指令周期，指令周期的长短与指令的复杂程度有关，一般是若干个 CPU 周期。时钟周期、机器周期和指令周期之间的关系如图 1-17 所示。

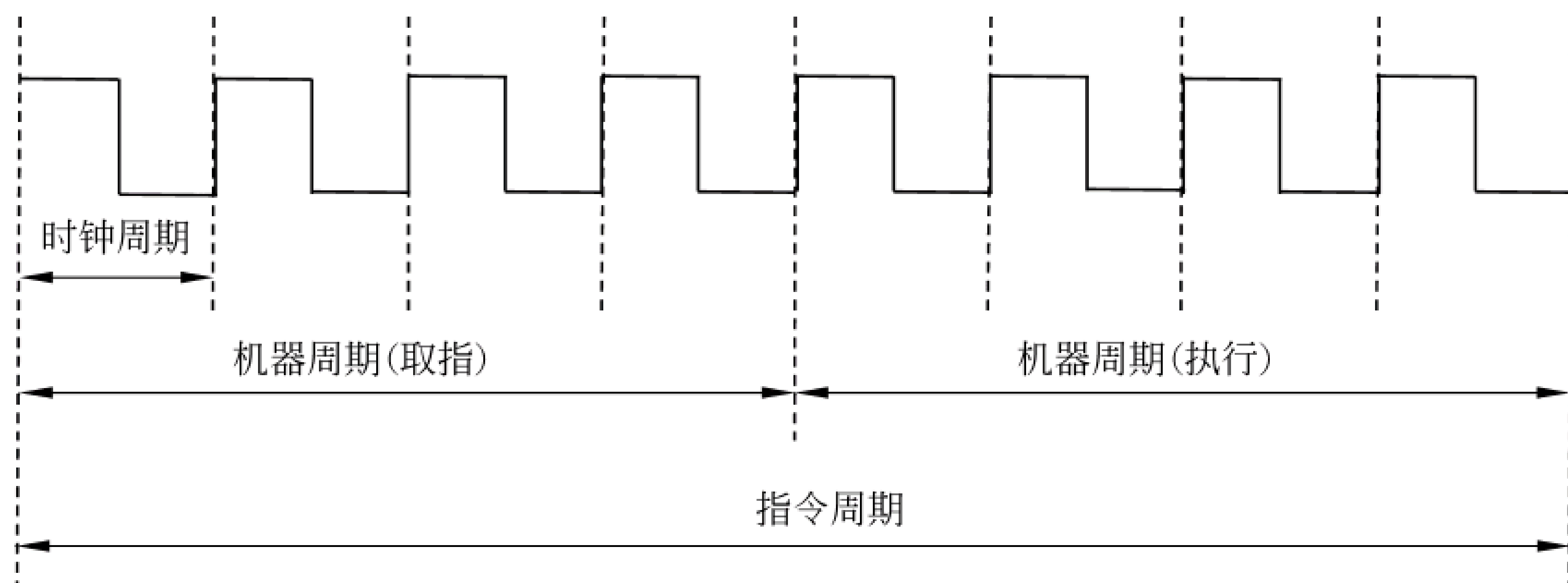


图1-17　指令周期、机器周期和时钟周期关系

在计算机执行指令的过程中,计算机内各部件的每一个动作都必须严格遵守时间规定,这些部件的协调是用时序信号来控制的,时序信号由时序发生器来产生。时序发生器是产生控制指令周期的时序信号的部件,当CPU开始取指令并执行指令时,操作控制器利用时序发生器产生的定时脉冲的顺序和不同的脉冲间隔,提供计算机各部件工作所需的各种定时控制信号,有条理、有节奏地指挥机器各个部件按规定时间动作。

1.3　嵌入式系统概述

当前,计算机技术已经进入后PC时代。人们使用的计算机系统已经不再只是工作于传统的桌面PC,很多的应用系统更多地工作在各种嵌入式平台上,如手机、智能洗衣机、智能手表以及数据采集器、智能定位装置和实时图像处理等各种嵌入式系统。

嵌入式系统是一种嵌入受控器件内部,为特定应用而设计的专用计算机系统。例如,智能洗衣机的智能洗涤程序及其平台构成的一个典型的嵌入式系统,该系统嵌入到洗衣机中并控制洗衣机的工作。区别于个人计算机系统,嵌入式系统通常执行的是带有特定要求和环境约束的任务,如要求极快的系统响应时间、较高的工作温度、极低的系统能耗和特别的安全性能。例如,应用于炼钢生产的天车定位装置,因为天车的快速移动要求具有极快的系统响应时间,而钢水辐射要求设备能工作在较高的环境温度。但是,个人计算机系统大多是通用计算机,具有通用的软硬件平台并安装有常用的应用软件,如现在大多数PC系统都是基于Wintel架构,即Microsoft公司的Windows操作系统与Intel公司的CPU所组成的个人计算机系统。

嵌入式系统一般针对一项特殊的任务,系统设计人员能够根据需要选择合适的硬件和软件平台,也可以减小系统尺寸或降低系统成本。实际上,很多嵌入式系统都会大量生产并广泛应用,嵌入式系统的成本就成为一个关键的设计问题。例如手机作为一种嵌入式系统,其价格作为未来市场推广成功与否一个关键要素。

嵌入式系统没有公认的明确定义,通常意义上,嵌入式系统可以描述为:以计算机技术为基础,面向特定应用,软件和硬件均可根据需要进行定制和裁剪,在系统可靠性、成本和功

耗等方面有着严格要求的专用计算机系统。

通用计算机系统和嵌入式系统相比较，嵌入式系统具有以下的显著特点：

1）专用性强

嵌入式系统大多面向特定的应用。因此，对嵌入式系统的硬件和软件系统都必须进行高效的设计和开发，尽可能去除不必要的冗余，保证面向应用的最为合适的运算能力。而且，在功耗、配置、内核处理能力、外围电路选择、系统响应时间和系统可靠性等方面具有明显的要求。例如，手机和智能手表等消费类产品要求具有良好的图形处理能力并有一定的容错能力，而航天军工和工业控制等工业类产品要求具有良好的实时计算能力且不允许出现错误，对系统的可靠性要求极高，也同时要求具有快速的系统响应。

2）实时性高

嵌入式系统对实时任务有很强的支持能力，能完成多任务并且有较短的中断响应时间。一般来说，需要优化中断控制器以及实时操作系统任务调度保障实时性。

3）种类繁多

嵌入式系统的多样性，集中体现为嵌入式微处理器和操作系统两个方面。

从嵌入式微处理器角度来说，已知的嵌入式微处理器大约有 1000 多种。嵌入式微处理器由通用计算机中的微处理器发展而来。与通用计算机的微处理器不同的是，在实际嵌入式应用中，只保留和嵌入式应用紧密相关的功能硬件，去除其他的冗余功能部分，因此其体积小、重量轻、功耗低、成本低及可靠性高。同时，嵌入式微处理器把 CPU、ROM、RAM 及 I/O 等元件以及各种外设集成到同一个芯片上，也称为单片机或微控制器。对于通用计算机系统来说，芯片基本上由 Intel 和 AMD 几家公司垄断，以 X86 和 AMD64 架构为主，操作系统软件方面，Microsoft 公司的 Windows 几乎占据了 90%的市场，可以说近代的通用计算机系统就是 Wintel 架构的垄断系统。但与全球 PC 市场不同的是，嵌入式微控制器约超过 1000 多种，体系架构有 ARM、MIPS、PowrPC、X86、68K 等 30 多个系列，以 32 位的嵌入式微控制器为例，Frescale、TI、ST、AVR、Atmel 等公司就有 100 种以上的嵌入式微控制器。从目前市场看，ARM 32 位微控制器架构占据垄断地位。

4）开发环境复杂

传统的通用计算机系统的开发环境与运行环境基本一致。例如，在一台计算机的 Windows 操作系统中开发一个学生管理系统，可能运行在另一台计算机的 Windows 操作系统上。

而对于嵌入式系统的开发来说，一般使用交叉开发模式，即将通用计算机与目标嵌入式系统进行连接，在通用计算机搭建开发环境，所开发的嵌入式代码“下载”到目标嵌入式系统进行运行。这种交叉开发模式无疑增加了系统开发的难度。

5）成本极其敏感

由于嵌入式系统往往大规模应用于各种生产、监测或消费等环境，往往使用量巨大。例如，遥控器、点菜机、温度监测设备、噪声监测设备甚至最近流行的智能手表、智能血糖仪等，都会有非常庞大的使用量。因此，每一个部件的成本都非常关键，总体成本或价格因素往往决定着产品的推广使用。

1.4 嵌入式系统架构

与通用计算机系统类似，嵌入式系统包括嵌入式硬件平台、嵌入式操作系统和嵌入式应用软件，如图 1-18 所示。

对于嵌入式系统来说，软件大多存储在只读存储器(ROM)中，一般不需要辅助存储器。

由于嵌入式系统面对的硬件种类多样，将开发者从繁琐的硬件细节中解放出来一直是嵌入式系统工业界面临的挑战。硬件抽象层(Hardware Abstraction Layer，HAL)是嵌入式系统开发中的一个重要中间件，硬件抽象层将微处理器的底层操作进行封装，隐藏了特定平台的硬件接口细节，为操作系统和应用软件提供虚拟硬件平台，使其具有硬件无关性，可在多种平台上进行移植。带有硬件抽象层的嵌入式系统结构，如图 1-19 所示。硬件抽象层的出现大大改进了嵌入式操作系统的通用性，硬件抽象层的定义一般由微处理器的生产厂家提供。

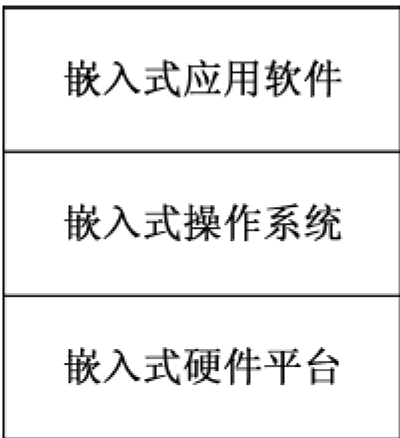


图 1-18 嵌入式系统组成

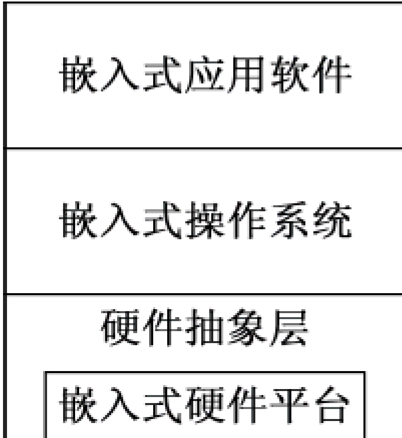


图 1-19 带硬件抽象层的嵌入式系统

板级支持包(Board Support Package，BSP)是介于主板硬件和操作系统中驱动层程序之间的一层，一般认为它属于操作系统一部分，主要是实现对操作系统的支持，为上层的驱动程序提供访问硬件设备寄存器的函数包，使之能够更好地运行于硬件主板。BSP 是相对于操作系统而言的，不同的操作系统对应于不同定义形式的 BSP，例如 VxWorks 的 BSP 和 Linux 的 BSP 相对于某一 CPU 来说尽管实现的功能一样，可是写法和接口定义是完全不同的。BSP 主要功能是屏蔽硬件细节，提供硬件驱动，具体功能包括：

(1) 硬件初始化，主要是 CPU 的初始化，为整个软件系统提供底层硬件支持；

(2) 为操作系统提供设备驱动程序和系统中断服务程序；

(3) 定制操作系统的功能，为软件系统提供一个实时多任务的运行环境；

(4) 初始化操作系统，为操作系统的正常运行做好准备。

嵌入式操作系统(Embedded Operating System，EOS)是指用于嵌入式系统的操作系统。嵌入式操作系统是一种用途广泛的系统软件，通常包括与硬件相关的底层驱动软件、系统内核、设备驱动接口、通信协议、图形界面、标准化浏览器等。嵌入式操作系统负责嵌入式系统的全部软、硬件资源的分配、任务调度，控制、协调并发活动。与 PC 操作系统相比，具有可剪裁、体积小、强调实时性、与应用程序紧密耦合等特点。目前在嵌入式领域广泛使用

的操作系统有：嵌入式实时操作系统 μC/OS-Ⅱ、嵌入式 Linux、Windows Embedded、VxWorks 等，以及应用在智能手机和平板电脑的 Android、iOS 等。在很多简单应用中，一般不需要操作系统，应用程序直接通过硬件抽象层对硬件资源进行控制。

如图 1-20 所示，嵌入式系统的硬件层中包含嵌入式微控制器、外扩存储器(SDRAM、ROM、Flash 等)、设备 I/O 接口等。在一片嵌入式微控制器基础上添加电源电路、时钟电路和存储器电路，就构成了一个嵌入式核心控制模块，操作系统和应用程序都可以固化在 ROM 中。

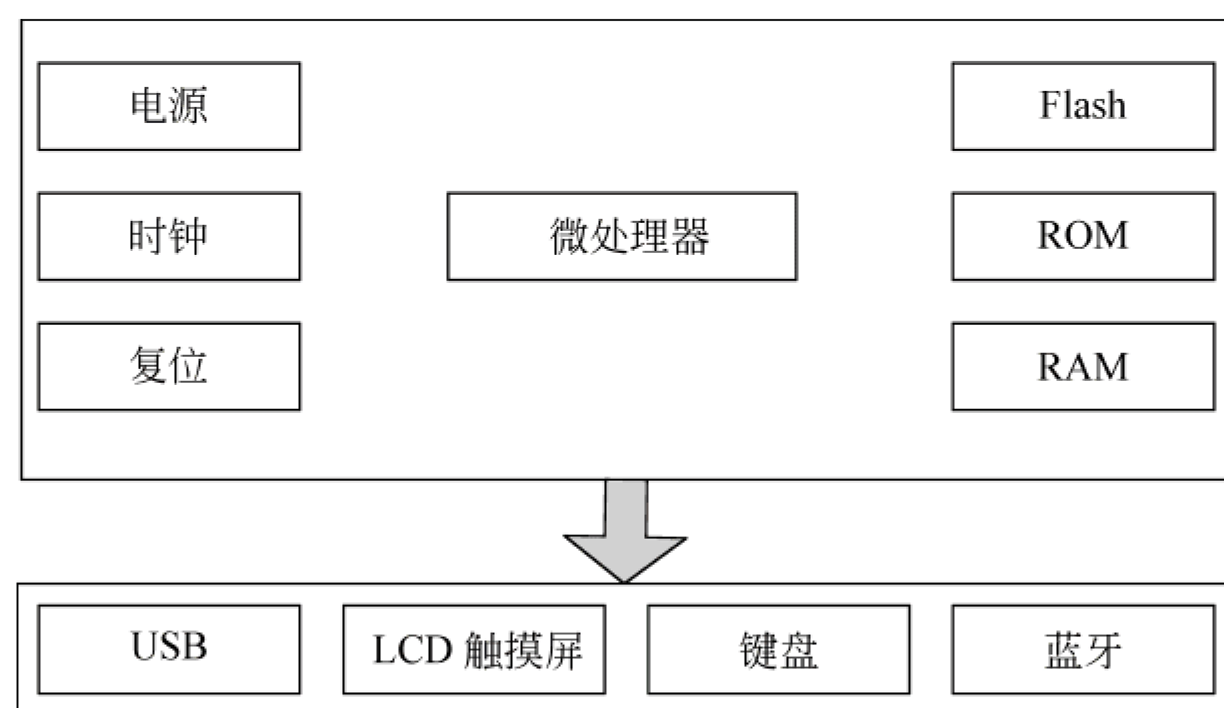


图 1-20　嵌入式系统的硬件组成

嵌入式微控制器是嵌入式系统的核心部件，一般采用 RISC 架构，嵌入式微控制器虽然在功能上和通用计算机的微处理器基本一致，但在可靠性、能耗、工作环境温度和抗电磁干扰等方面做了很多增强内部集成了 ROM、RAM 和大量外设接口，对于简单的应用，无需外扩存储，单片微控制器即可构成最小系统。

1.5　嵌入式系统的典型应用

1）工业控制

基于嵌入式芯片的工业自动化设备已经大量地应用于实际生产。很多 8 位、16 位和 32 位嵌入式微控制器应用于工业过程监控，如工业生产过程控制、数字机床、电力系统、电网设备监测和石油化工生产等，大大地减少了人力资源投入。就传统的工业控制产品而言，低端型采用的往往是 8 位单片机。随着嵌入式技术的发展，32 位甚至 64 位的微处理器逐渐成为工业控制设备的核心，在未来几年内必将获得长足的发展。

一款基于 ARM9 内核和 Windows CE 操作系统的工业计算机，如图 1-21 所示。该产品以 ARM9 低功耗嵌入式 CPU 为核心，主频为 400MHz，嵌入式操作系统采用 Windows CE 6.0，提供高性能嵌入式人机界面。

2）汽车电子

嵌入式系统在汽车和交通行业的应用主要表现为车辆导航、流量控制、信息监测和汽车

服务等。内嵌 GPS/北斗和 GSM、3G/4G 模块的移动定位终端已经在各种运输行业获得了成功的使用。由于 GPS 设备价格低廉，已经从尖端产品进入了普通百姓的家庭。一款基于 ARM11 内核的导航仪，如图 1-22 所示。该导航仪拥有分辨率 800×480 的 7 英寸高清数字屏，采用 ARM11 内核，主频 600MHz，内置 GPS，标配 4G SD(Secure Digital Memory Card)卡，支持 16G U 盘、32G SD 卡及硬盘扩展。



图 1-21　基于 ARM9 内核和 Windows CE 的工业计算机



图 1-22　基于 ARM11 内核的导航仪

3）信息家电

信息家电是嵌入式系统最大的应用领域。冰箱、空调、洗衣机和电饭煲等家用电器的网络化和智能化将引领人们的生活步入一个崭新的空间。即使用户不在设备身边，也可通过网络进行远程控制。一款基于 ARM Cortex 内核和 Android 操作系统的电视盒，如图 1-23 所示。该款电视盒采用当今移动互联设备上 ARM Cortex-A9 1.2 GHz 核心处理器，内置 3D 图形处理器 Mali-400，搭配 512MB DDR3(Double Data Rate)大容量内存，性能超强，相当于通用计算机的高速运算能力，具有流畅的高清视频和游戏体验。同时采用 Google Android 4.0 操作系统，不但在系统稳定性有了进一步的保证，还可以随意安装使用 Google Market 数百万计的应用程序和游戏。



图 1-23　基于 ARM Cortex 内核和 Android 操作系统的电视盒

4）环境监测

在很多环境恶劣、地况复杂的地区，嵌入式系统将实现无人值守 24 小时不间断监测，如水文资料实时监测、水土质量监测、地震监测、实时气象信息监测、水源和空气污染监测等。

一款基于 ARM 内核的物联网监测系统，如图 1-24 所示。搭配各种传感器的 ZigBee 节点用于测量温度、湿度、光照或气体成分等，ARM9 内核的通信网关支持 RS-232、USB 和

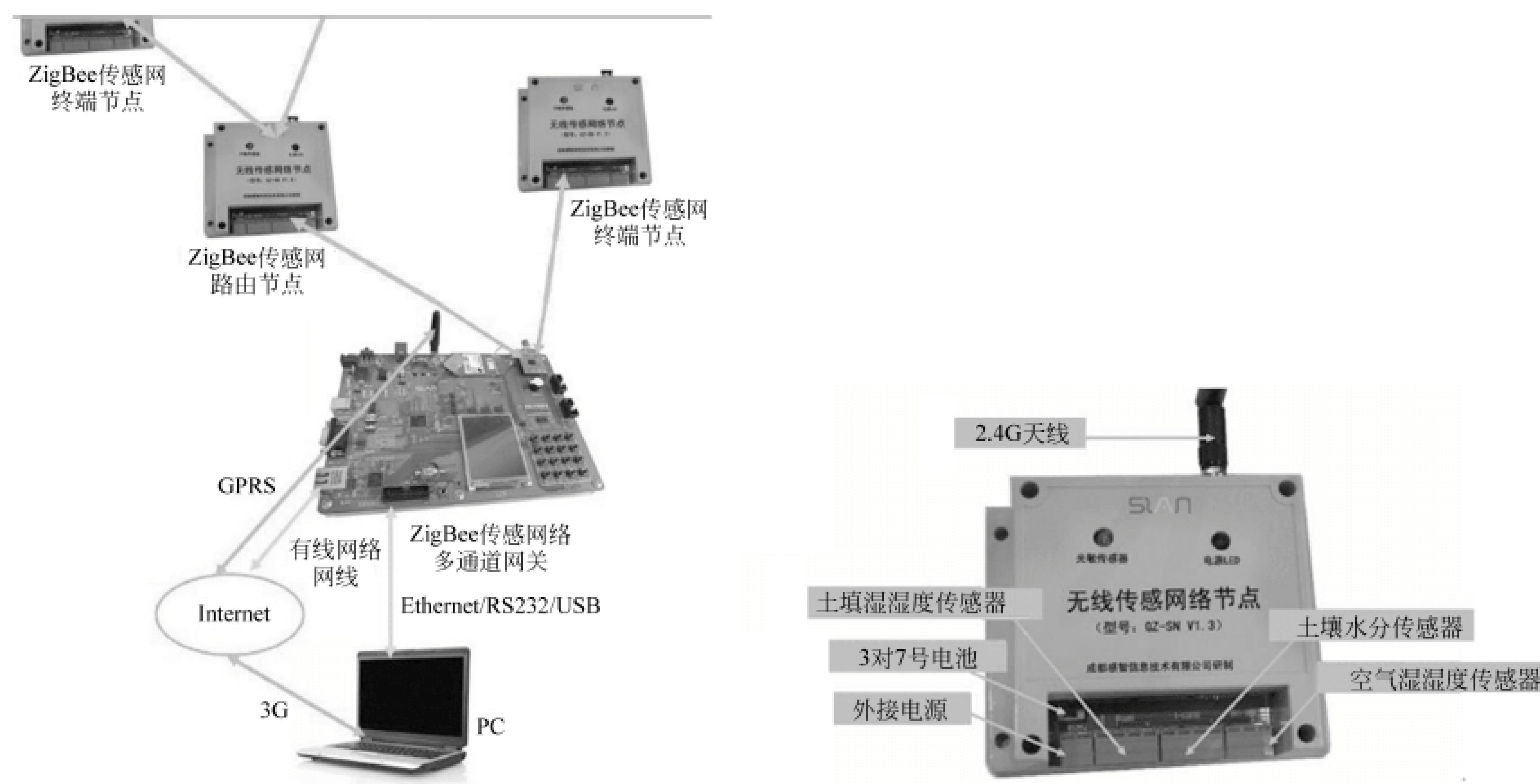


图 1-24　基于 ARM 内核的物联网监测系统

GPRS 等各种通信方式与上位机监控系统进行互联。

5）健康管理

嵌入式系统已经应用到健康管理的方方面面，如智能血压计、智能脉搏计和智能血糖仪等。嵌入式系统在健康管理方面的应用，必将成为未来最具有前景的应用领域之一。一款支持 iOS 或 Android 操作系统智能血压计，如图 1-25 所示。智能血压计可通过蓝牙连接智能手机传输数据，并生成图表，让用户更好地了解自己的血压状况。该智能血压计设计十分简洁，支持 iOS 及 Android 设置。不需要任何专业的医疗技巧，只需将尼龙腕带绑在手臂上，单击开始按键，就可以监测血压。



图 1-25　支持 iOS 或 Android 操作系统智能血压计

6）机器人

嵌入式芯片的发展将使机器人在微型化、智能化方面的优势更加明显，同时会大幅度降低机器人的价格，使其在工业领域和服务领域获得更为广泛的应用。一款基于 ARM 内核和 μC/OS-Ⅱ 操作系统的家庭机器人，如图 1-26 所示。该机器人集红外、超声波、湿度、温度、烟雾和煤气等多种传感器于一体，具有对外无线通信的功能，使用了嵌入式领域应用广

泛、处理能力极强的32位ARM处理器，软件采用实时性操作系统μC/OS-Ⅱ，来协调机器人自身复杂的机械控制及处理周围复杂未知的环境因素。

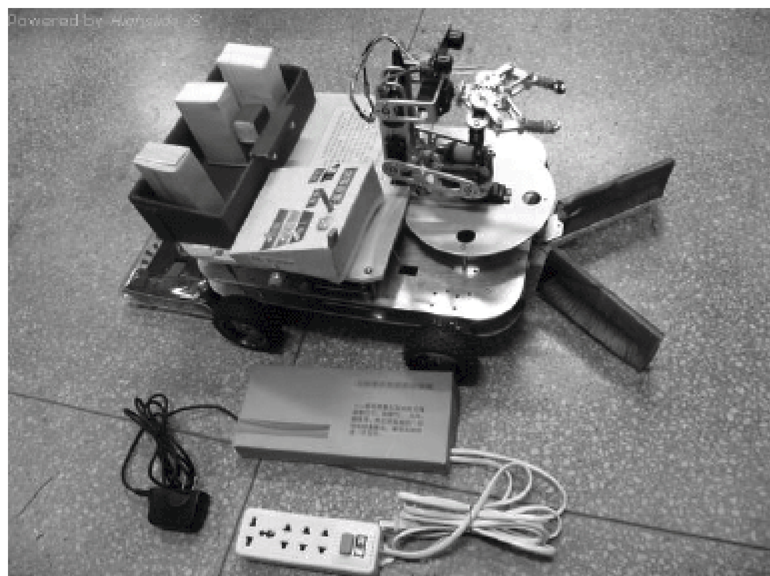


图1-26　基于ARM内核和μC/OS-Ⅱ操作系统的家庭机器人

1.6　典型嵌入式开源硬件和软件系统

1.6.1　开源硬件平台

由于嵌入式产品都有相似的微处理器内核及通用外围功能单元，将它们集合起来，可做成一个供众多嵌入式产品个性化开发的产品平台。最引人注目的是由树莓派(Raspberry Pi)引发的智能化通用板级开源硬件。

最早推出的Raspberry Pi是为学习计算机编程而设计的一个只有信用卡大小的板级微型电脑，配置了Linux操作系统。由于低价位、功能强大、有众多的外围电路与I/O端口、易开发的软件配置，树莓派迅速成为板级开源硬件的理想化通用产品平台。到目前为止，已出现了香蕉派、Arduino、BeagleBoard等一系列开源硬件平台，其中三大主流平台位Arduino、BeagleBoard和Raspberry Pi，它们已建立了完整的硬件、软件生态。

1）树莓派

树莓派(Raspberry Pi)由英国的树莓派基金会所开发，目的是以低价硬件及自由软件刺激学校的基础计算机科学教育，如图1-27所示。树莓派配备一枚700MHz博通出产的ARM架构BCM2835处理器，256MB内存(B型已升级到512MB内存)，使用SD卡当作存储媒体，且拥有一个Ethernet，两个USB接口，以及HDMI(支持声音输出)和RCA端子输出支持。操作系统采用开源的Linux系统，比如Debian、ArchLinux，自带的Iceweasel、KOffice等软件能够满足基本的网络浏览、文字处理以及计算机学习的需要。树莓派基金会提供了基于ARM架构的Debian、Arch Linux和Fedora等的发行版供大众下载，以Python作为主要编程语言，支持BBC BASIC、C语言和Perl等编程语言。树莓派基金会于2016年2月发布了树莓派3，较前一代树莓派2，树莓派3的处理器升级为了64位的博通BCM2837，并首次加入了Wi-Fi无线网络及蓝牙功能，而售价仍然是35美元，目前已发布

树莓派 4。

2）Arduino

Arduino 是一个开放源代码的单芯片微计算机，由一个欧洲开发团队于 2005 年冬季开发。它使用了 Atmel AVR 单片机，采用了基于开放源代码的软硬件平台，构建于开放源代码 simple I/O 接口板，并且具有使用类似 Java、C 语言的 Processing/Wiring 开发环境。

Arduino 能通过各种各样的传感器来感知环境，通过控制灯光、马达和其他的装置来反馈、影响环境。Arduino 包括一个硬件平台（Arduino Board）和一个开发工具（Arduino IDE）。两者都是开放的，既可以获得 Arduino 开发板的电路图，也可以获得 Arduino IDE 的源代码。Arduino Board 提供了基本的接口和 USB 转串口模块，如图 1-28 所示。使用者只需要用一个 USB 线就可以连接计算机和 Arduino Board，完成编程和调试。Arduino 使用一种简单的专用编程语言，使用者不必掌握汇编语言和 C 语言等复杂技术就可以进行开发。IDE 可免费下载，并开放源代码，跨平台，极为便利。

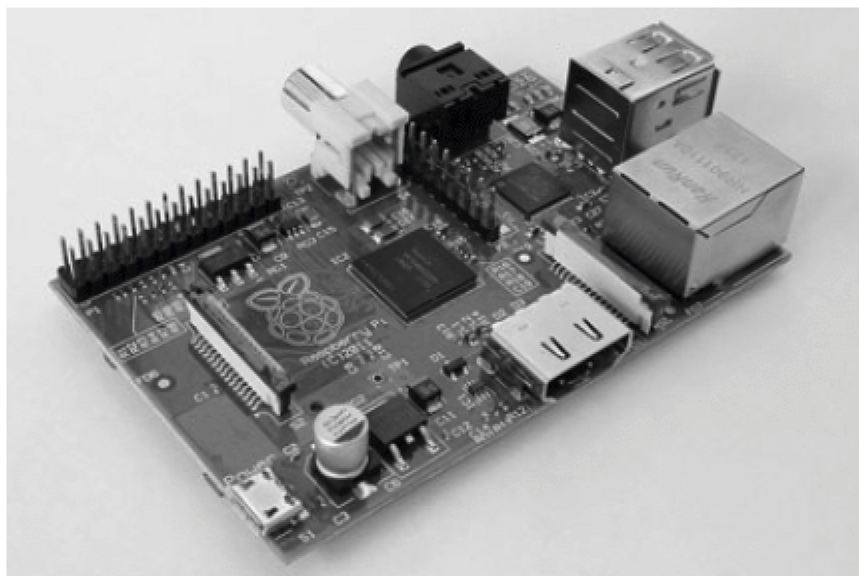


图 1-27　树莓派 B2

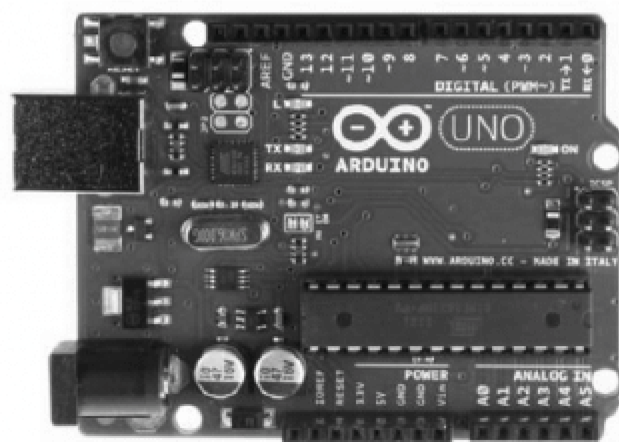


图 1-28　Arnduino 开发板

3）BeagleBoard

BeagleBoard 是开源硬件领域知名社区 BeagleBoard.org 推出的、全球第一款开源的 ARM 开发板，如图 1-29 所示。不同于热门的开源平台 Arduino，BeagleBoard 的功能更强大、应用更复杂。BeagleBoard 跨越了台式机和嵌入式计算机的界限，同时与开源社区展开创建全新应用的协作，为开源社区提供成本更低、更新、更出色的开发平台。BeagleBone 是 BeagleBoard 的升级版本，只集成了一些必不可少的接口功能，如 USB、以太网口。继 BeagleBone 之后，德州仪器推出的 BeagleBone Black，采用 TI 最新 Cortex-A8 架构 Sitara 处理器，主频可提升至 1GHz。

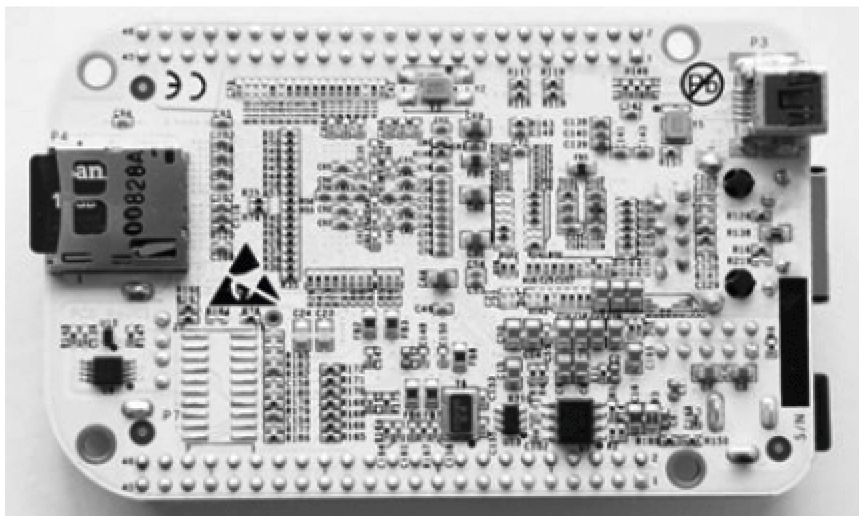


图 1-29　BeagleBone

1.6.2 嵌入式开源操作系统

嵌入式操作系统是嵌入式应用系统的核心软件平台，目前常见的嵌入式操作系统有：eCos、μC/OS、VxWorks、pSOS、Nucleus、ThreadX、Rtems、QNX、INTEGRITY、OSE、C Executive、CMX、SMX、emOS、Chrous、VRTX、RTX、FreeRTOS、LynxOS、ITRON、Symbian、RT-thread，以及 Linux 家族的各种版本（如 μClinux、Android 等），还有微软家族的 WinCE、Windows Embedded、Windows Mobile 等。

1）Linux

Linux Torvalds 在 1991 年发表的 Linux 开放操作系统，是由互联网上的志愿者们开发的，它吸引了许许多多忠实的追随者。自 1999 年稳定的 2.2 版本发布以来，Linux 不仅已经在服务器和台式机上取得了巨大的成功，也正在嵌入式系统中大放异彩。许多人认为，Linux 之所以获得嵌入式市场的广泛认可，关键是得益于 Linux 极高的质量和极强的生命力。当然，能够给 Linux 开发人员提供充分的灵活性和开放的源码，不收取运行许可使用费也是开发者选择 Linux 的极好理由。与商业软件授权方式不同的是，开发者可以自由地修改 Linux，能最大地满足他们的应用需要。在技术上，因为基于 UNIX 技术，Linux 提供广泛的功能强大的操作系统功能，包括内存保护、进程和线程，以及丰富的网络协议。Linux 与 POSIX 标准兼容，从而提高了应用的可移植性。Linux 支持多种微处理器、总线架构和设备，通常情况下，芯片公司的驱动程序、应用相关的中间件、工具和应用程序都是先为 Linux 开发，后来才移植到其他 OS 平台的。这些特性都非常适合于嵌入式系统应用。

2）MontaVista Linux

MontaVista Linux 不只是一个通用的 Linux 发行版，更是为嵌入式系统所需的可靠性和实时性（通过对 2.4 内核加入实时补丁）而精心设计的，支持高端嵌入式系统使用的处理器架构 x86、ARM、PowerPC 和 MIPS，以及一系列的驱动程序和板级支持包。它有一整套的开发工具、闪存和固态存储文件系统，还有很容易监视系统完整性和性能的各种工具。MontaVista Linux 在通信基础设备、智能手机、数字电视机和机顶盒等各种嵌入式系统中得到广泛应用。

3）eCos

eCos 全称是 Embedded Configurable Operating System，它诞生于 1997 年，eCos 最大的特点是模块化，内核可配置。它是一个针对 16/32/64 位处理器的可移植开放源代码的嵌入式 RTOS。eCos 提供的 Linux 兼容的 API 能让开发人员轻松地将 Linux 应用移植到 eCos。eCos 的核心具备一般 OS 功能，如驱动和内存管理、异常和中断处理、线程的支持，还具备实时操作系统的特点，如可抢占、最小中断延迟、线程同步等。eCos 支持大量外设、通信协议和中间件，如以太网、USB、IPv4/IPv6、SNMP、HTTP 等。

4）Android

Android 是谷歌公司开发的一个针对高端智能手机的操作系统。其实 Android 不仅仅是一个操作系统，也是一个软件平台，可以应用在更加广泛的设备中。在实际应用中，

Android 是一个在 Linux 上的应用架构，优势是能够帮助开发者快速地布置应用软件。Android 的开发主要还是集中在移动终端上，在其他的市场上 Android 也潜力巨大。比如智能电视、消费电子产品、通信、汽车电子产品、医疗仪器和智能家居应用等。

5）μC/OS-Ⅱ

μC/OS-Ⅱ由 Micrium 公司提供，是一个可移植、可固化、可裁剪、抢占式多任务实时内核，它适用于多种微处理器、微控制器和数字处理芯片（已经移植到超过 100 种以上的微处理器应用中）。该系统源代码开放、整洁、注释详尽。μC/OS-Ⅱ可管理多达 63 个应用任务，并可以提供如下服务：信号量、互斥信号量、事件标识、消息邮箱、消息队列、任务管理、固定大小内存块管理、时间管理另外，在 μC/OS-Ⅱ内核之上还可以选增、μC/FS 文件系统模块、μC/GUI 图形软件模块、μC/TCP-IP 协议栈模块、μC/USB 协议栈模块等。

6）μCLinux

μCLinux 表示 micro-control Linux. 即“微控制器领域中的 Linux 系统”，是 Lineo 公司的主打产品，同时也是开放源码的嵌入式 Linux 的典范之作。μCLinux 主要是针对目标处理器没有存储管理单元 MMU（Memory Management Unit）的嵌入式系统而设计的，是唯一可以在低端 MCU 上运行的 Linux，可以在特定的 Cortex-M3、M4 和 M7 等型号上运行。μClinux 的 RAM 和 ROM 资源需求较多，需要 MCU 内置存储器控制器，使用外部扩展 DRAM 芯片来满足内存要求。现在 μClinux 已被并入到主线 Linux 内核中。

7）FreeRTOS

FreeRTOS 这是一个开源的项目，属于轻量级内核，API 比较全，支持 AVR、ARM、MSP430 等处理器，同时有移植好的 TCP/IP 协议栈 μIP。

8）ARM Mbed

ARM 面向物联网的操作系统针对小巧、电池供电的物联网端点，这些端点在 Cortex-M 系列 MCU 上运行，可能只有 8KB 内存。mbend 提供了多线程和实时操作系统支持，在设计当初就针对无线通信，可通过 Mbed Device Connector 来安全地提取数据的云服务。

除以上开源嵌入式操作系统以外，微软的 Windows CE、Windows Phone、苹果的 iOS 都是典型主流嵌入式操作系统，老牌的嵌入式操作系统代表为风河公司的 VxWorks。

VxWorks 是由支持多核、32/64 位嵌入式处理器、内存管理的 Vxworks、workbench 开发工具（包括多种 C/C++ 编译器和调试器）、连接组件（USB、IPv4/IPv6、多种文件系统等）、网络协议和图像多媒体等模块组成。除了通用平台外，VxWorks 还包括支持工业、网络、医疗和消费电子等的特定平台产品。风河公司的 VxWorks 以其高可靠性和优异的实时性被广泛应用在通信、军事、航空航天、工业控制等领域。比如在美国的 F-16、FA-18 战斗机、B-2 隐形轰炸机和爱国者导弹上都有使用，最为著名的是 1997 年 4 月在火星表面登陆的火星探测器、2008 年 5 月登陆的凤凰号和 2012 年 8 月登陆的好奇号火星车，也都使用到了 VxWorks。

Chapter 2

第 2 章　Cortex-M3 微处理器的体系结构

【导读】 ARM Cortex-M3 是目前低成本嵌入式系统使用最为广泛的 CPU 内核，本章首先回顾 ARM 微处理器的发展，重点介绍 Cortex-M3 处理器的内核结构、工作模式、存储映射等基本原理，然后以 ST 公司的 STM32L152 为例，对该微控制器的结构、引脚说明、时钟控制、存储等进行详细介绍，为指令系统和后续 I/O 接口的学习奠定基础。

2.1　ARM 微处理器系列介绍

ARM 的全拼是 Advanced RISC Machines，中文的意思为先进的精简指令集机器，是一家总部位于英国剑桥的半导体微处理器公司。目前，ARM 在手机处理器市场占据了超过 90％的份额，在平板电脑处理器市场占据了超过 80％的份额。

表 2-1 为 ARM 的系列处理器代号及其处理器架构版本。ARM 处理器大约有 6 个流行的产品系列，分别为 ARM7、ARM9、ARM10、ARM11、SecureCore 和 Cortex。其中，ARM7、ARM9、ARM10 和 ARM11 是早期的处理器命名方式，每个系列提供可配置的不同性能的处理器版本；SecureCore 系列主要面向安全设备设计；ARM11 之后，所有处理器均以 Cortex 系列命名，根据不用的市场包括三个子系列，分别是 Cortex-A、Cortex-M 和 Cortex-R。其中，Cortex-A 系列面向复杂操作系统和用户应用，支持 ARM、Thumb 和 Thumb-2 指令集；Cortex-R 系列面向嵌入式或实时系统，也支持 ARM、Thumb 和 Thumb-2 指令集；Cortex-M 系列面向成本敏感的嵌入式应用，只支持 Thumb-2 指令集。

ARM 系列处理器有三种指令集：

(1) ARM 指令集：ARM 指令集为 32 位指令集，可以实现 ARM 架构下的所有功能。

(2) Thumb 指令集：Thumb 指令集是针对代码存储密度的需求对 32 位 ARM 指令集的改进，其目标是实现更高的代码密度。具体做法是将 32 位 ARM 指令集的部分指令压缩成 16 位的编码方式，而当指令执行时，再解码成相应的 32 位 ARM 指令功能。这种经过压缩-解码方式，以牺牲部分处理器性能换取了代码密度的提升。相对于 ARM 指令集，Thumb 指令集在代码密度方面大约提升了 30％，但 Thumb 不是一个完整的指令集，部分 32 位指令无法压缩，Thumb 需要和 ARM 指令集配合使用，两种指令执行时处理器需要在不同的状态切换。

(3) Thumb-2 指令集：Thumb-2 指令集是在 Thumb 指令集的基础上发展而来 16 位/

32位混合指令集，增加了一些16位Thumb指令来改进程序的执行流程，增加一些新的32位Thumb指令来实现ARM指令的专有功能，因此，Thumb-2指令集中有两类不同长度的指令，不兼容ARM指令集，并且采用了新方法实现16位和32位指令的执行，无需进行Thumb和ARM指令的压缩解压转换，也无需在ARM和Thumb状态进行来回切换，大大提升了运算效率。

Cortex-M系列是基于ARMv7架构。ARMv7架构是在ARMv6架构的基础上发展而来。ARMv7架构采用Thumb-2技术，Thumb-2技术比纯32位代码少使用大约31%的内存，却比基于Thumb技术的代码在性能上提高大约38%。ARM处理器代号与指令集架构之间的关系如表2-1所示。

表2-1　ARM处理器与对应的架构

ARM处理器内核代号	架　构
ARM1	ARMv1
ARM2	ARMv2
ARM2As，ARM3	ARMv2a
ARM6，ARM600，ARM610，ARM7，ARM700，ARM710	ARMv3
StrongARM，ARM8，ARM810	ARMv4
ARM7TDMI，ARM710T，ARM720T，ARM740T，ARM9TDMI，ARM920T，ARM940T	ARMv5T
ARM9E-S，ARM10TDMI，ARM1020E	ARMv5TE
ARM1136J(F)-S，ARM1176JZ(F)-S，ARM11，MPCore	ARMv6
ARM1156T2(F)-S	ARMv6T2
ARM Cortex-M，ARM Cortex-R，ARM Cortex-A	ARMv7、ARMv8

第一款采用ARMv7-M架构的处理器架构是Cortex-M3，随后ARM针对嵌入式市场的需求，形成了系列M处理器架构。

Cortex-M0、M0+、M1系列：Cortex-M0是目前最小的ARM处理器，该处理器支持ARMv6-M架构，芯片面积非常小，能耗极低，且编程所需的代码占用量很少，这就使得开发人员可以直接跳过16位系统，以接近8位系统的成本开销获得32位系统的性能。Cortex-M0+是以Cortex-M0处理器为基础，保留了全部指令集和数据兼容性，同时进一步降低了能耗，提高了性能，两级流水线，性能效率可达1.08DMIPS/MHz。Cortex-M1是第一个专为FPGA中的实现设计的ARM处理器。

Cortex-M3：ARMv7-M架构，改为3级流水哈佛结构，是目前M系列中应用最广泛的处理器架构。Cortex-M4在Cortex-M3基础上增加了DSP支持和单精度浮点运算加速，用以满足需要有效且易于使用的控制和信号处理功能混合的数字信号控制市场。Cortex-M7具有六级流水线、灵活的系统和内存接口、缓存(Cache)、DSP和双精度浮点运算加速。Cortex-M系列核心的比较如表2-2所示。

表 2-2　Cortex-M 系列核心的对比

类　　别	Cortex-M0	Cortex-M3	Cortex-M4	Cortex-M7
体系结构	ARMv6-M	ARMv7-M	ARMv7-M	ARMv7-M
ISA 指令集	Thumb，Thumb-2	Thumb，Thumb-2	Thumb，Thumb-2	Thumb，Thumb-2
DSP 扩展	—	—	支持	支持
浮点单元	—	—	单精度浮点	双精度浮点
DMISP 性能	0.9	1.25	1.25	2.5
内存保护	—	MPU	MPU	MPU

2.2　ARM Cortex-M3 体系结构

Cortex-M3 微处理器是一个高性能的 32 位处理器，主要面向微控制器市场。它集成了名为 CM3Core 的中央处理器内核和高效的总线，实现了内置的中断控制、存储器保护以及系统的调试和跟踪功能。具有快速中断处理机制、高效的内核运算性能和多种低功耗睡眠模式，支持 Thumb-2 指令集，确保较高的代码密度和较低的存储要求。

2.2.1　总体架构

Cortex-M3 处理器的内部架构如图 2-1 所示，其主要包括 5 个功能部件。

(1) 处理器内核 CM3Core，Cortex-M3 处理器的中央处理核心。

(2) 中断控制器(Nested Vectored Interrupt Controller，NVIC)。NVIC 是一个在 Cortex-M3 中内建的中断控制器，支持中断嵌套，采用向量中断机制，在中断发生时，它会自动取出对应的中断服务例程入口地址，并且直接调用，无需软件判定中断源，由此缩短了中断延时。

(3) 总线矩阵 BusMatrix，总线矩阵是 Cortex-M3 内部总线系统的核心，它是一个 AHB 总线互连网络，可以让数据在不同的总线之间并行传送(相当于网络交换机功能)。

(4) 存储保护单元 MPU(可选单元)，其主要功能是把存储器分成不同区域分别予以保护，让某些区域在用户模式下变成只读，特权模式下可读写，从而实现关键数据的保护。

(5) 处理器跟踪和调试接口，包括 FPB(Flash 修补和断点单元)、DWT(数据观察点和触发单元)、ITM(指令跟踪宏单元)、ETM(嵌入式跟踪宏单元)和 TPIU(跟踪端口接口单元)和一个串行线调试端口(SW-DP)/串口线 JTAG 调试端口(SWJ-DP)。ETM、TPIU、SW/JTAG-DP 和 ROM 表是可选的。

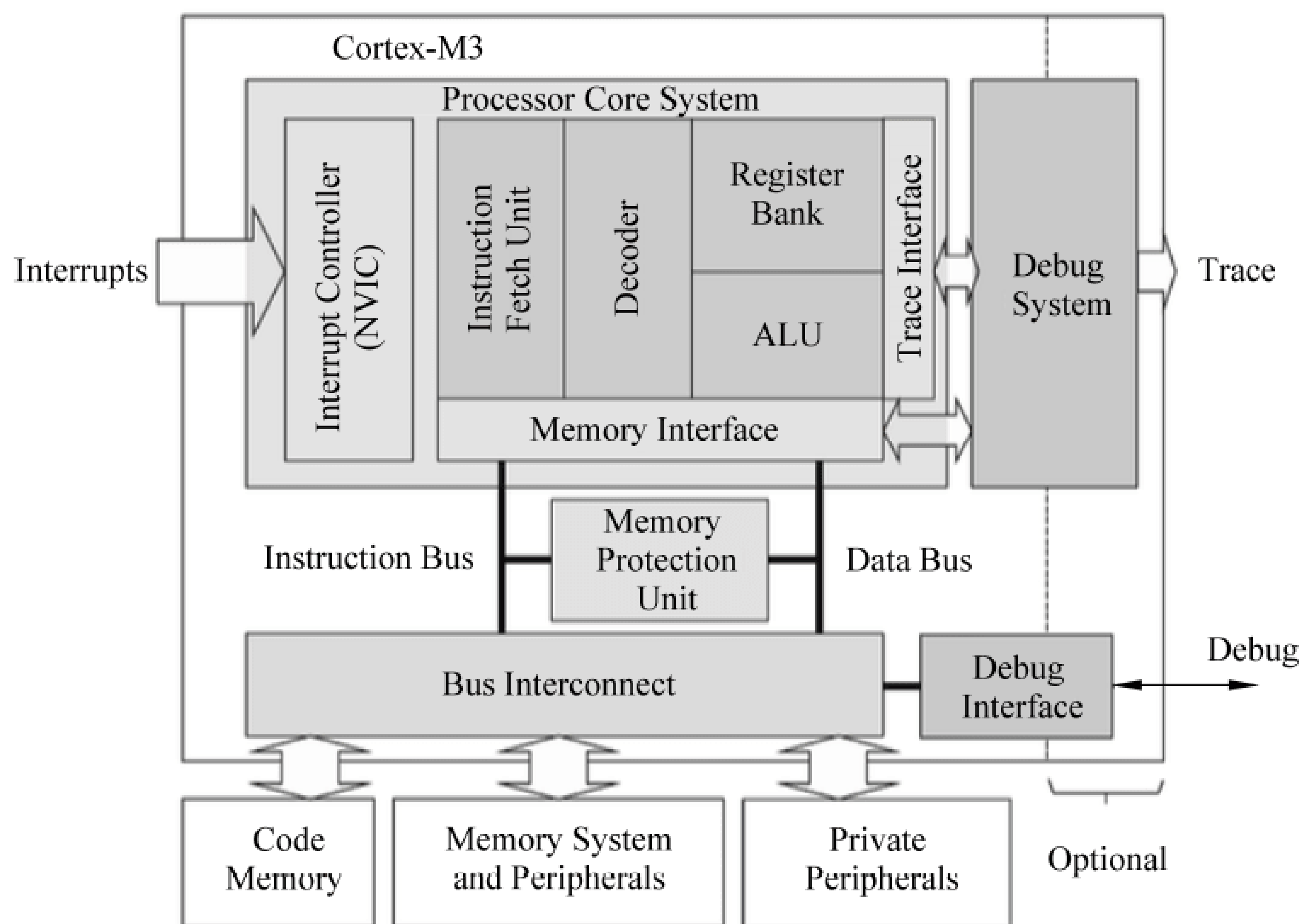


图 2-1　Cortex-M3 处理器内核架构

1．处理器内核

Cortex-M3 中央内核采用哈佛架构，指令和数据各使用一条总线。内核流水线分 3 个阶段：取指、译码和执行。当遇到分支指令时，译码阶段也包含指令预取，提高执行速度。Cortex-M3 内核包含 Thumb 和 Thumb-2 指令译码器、支持硬件乘法和除法的 ALU、控制逻辑和用于连接处理器其他部件的接口。Cortex-M3 处理器是一个 32 位处理器，带有 32 位宽的数据路径，寄存器和存储器接口。其中有 13 个通用寄存器，两个栈指针，一个链接寄存器，一个程序计数器和一系列包含编程状态寄存器的特殊寄存器。Cortex-M3 处理器支持两种工作模式（线程模式和异常处理模式）和两个等级的访问形式（有特权或无特权），实现对操作系统安全保护的支持。

2．NVIC 中断控制器

Cortex-M3 集成了一个可配置的嵌套向量中断控制器 NVIC。NVIC 支持 256 个中断和 256 种中断优先级，与处理器核心紧密耦合，提供中断服务程序（Interrupt Service Routine，ISR）的快速执行。其主要特性包括：

（1）CPU 内部占用 16 个中断，提供给处理器厂家的外部中断可配置为 1～240 个，中断可屏蔽，同时也支持一个不可屏蔽中断（Non-Maskable Interrupt，NMI）。

（2）优先级的种类可配置，支持最少 8 种，最多 256 种不同的优先级，支持中断嵌套。

（3）支持优先级分组，中断优先级可动态地重新配置。

（4）支持末尾连锁（tail-chaining）和迟到（late arrival）中断技术，提高响应速度。

（5）处理器状态在进入中断时自动保存，在保存状态的同时从存储器中取出异常向量，实现更加快速地进入 ISR（中断服务程序），中断返回时自动恢复，无需多余的指令。

3. 总线矩阵

总线矩阵用于配置Cortex-M3处理器核心和系统总线、存储器以及调试单元之间的总线连接，其支持的总线包括：

(1) ICode总线，该总线用于从代码空间取指令和向量，是32位AHBLite总线。

(2) DCode总线，该总线用于对代码空间进行数据加载/存储以及调试访问，是32位AHBLite总线。

(3) AHB系统总线，该总线用于对系统空间执行取指令和向量，数据加载/存储以及调试访问，是32位AHBLite总线。

(4) PPB私有外围设备总线，该总线用于对PPB空间(处理器集成的调试单元)进行数据加载/存储以及调试访问，是32位APB总线。

2.2.2 操作模式

Cortex-M3处理器有两种工作状态：

(1) Thumb-2状态：Thumb-2指令的正常执行状态。

(2) 调试状态：处理器停机调试时进入该状态。

Cortex-M3处理器支持两种工作模式：线程模式和异常处理模式。

(1) 线程模式(Thread Mode)：处理器工作在线程模式，用于执行应用程序，在复位时处理器进入线程模式，异常返回时也会进入该模式。

(2) 异常处理模式(Handler Mode)：处理器工作在异常处理模式，用于处理异常事件和中断。出现异常时处理器进入异常处理模式，当完成了异常处理之后，处理器将返回到线程模式。

Cortex-M3处理器的程序代码运行级别分为特权级执行和非特权级执行(也称用户级执行)。

非特权级别执行时对有些资源的访问受到限制或不允许访问(如CPS指令、系统控制空间的大部分寄存器等)，特权级别执行可以访问所有资源。当处理器工作在异常处理模式下时始终是特权访问，工作在线程模式下可以是特权访问，也可以是非特权访问。

Cortex-M3在线程模式下，控制寄存器CONTROL用于决定处理器处于特权级别还是非特权级别。程序在特权级别下可通过MSR指令将CONTROL寄存器的最低位CONTROL[0]置0，配置为非特权(用户)访问，但在非特权访问级别，本身不能主动回到特权访问。

【思考题：如何让用户级的程序主动进入特权级?】

根据Cortex-M3处理器的工作模式和特权等级，可以将Cortex-M3处理器划分为三种工作状态：特权级异常处理模式、特权级线程模式和非特权级线程模式，工作状态之间的转换，如图2-2所示。

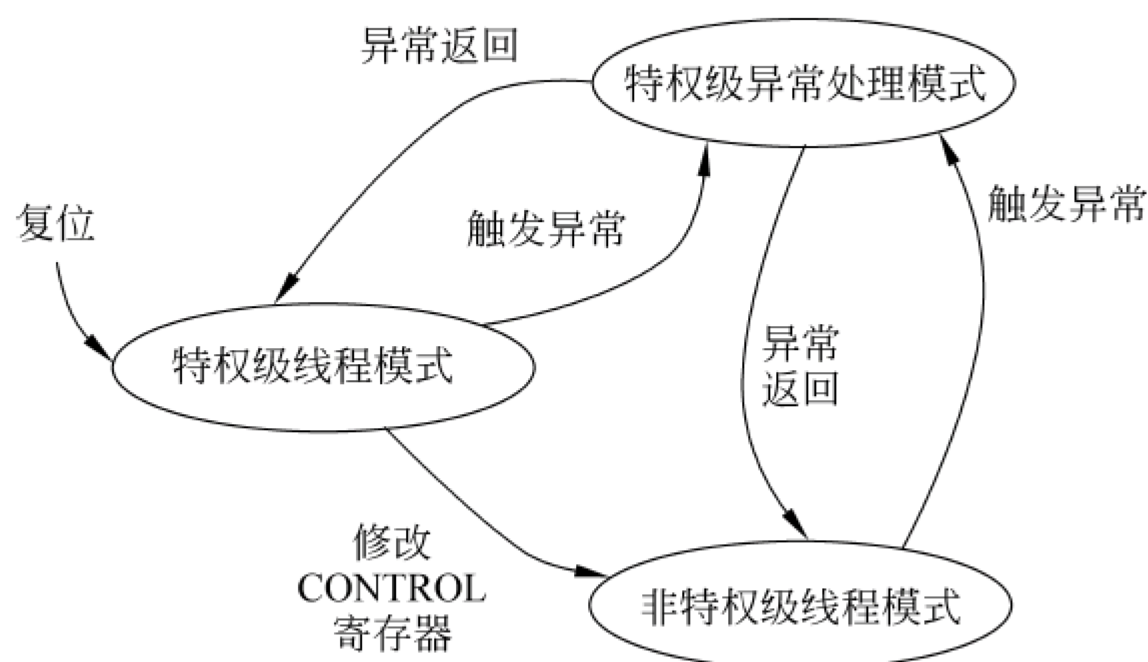


图 2-2 处理器工作状态的切换

由图 2-2 可见，只有处于特权级才可以通过改写控制寄存器 CONTROL 来改变处于线程模式的程序的特权等级，即可以由特权级线程模式直接过渡到非特权级线程模式，反之，则不能由非特权级线程模式通过修改控制寄存器 CONTROL 直接过渡到特权级线程模式。处于非特权线程模式的程序，只有进入异常处理模式才具有特权执行权限，此时可以通过修改 CONTROL 寄存器让程序在中断服务执行完成后回到特权级线程模式。

三种状态之间的切换示例如图 2-3 所示。

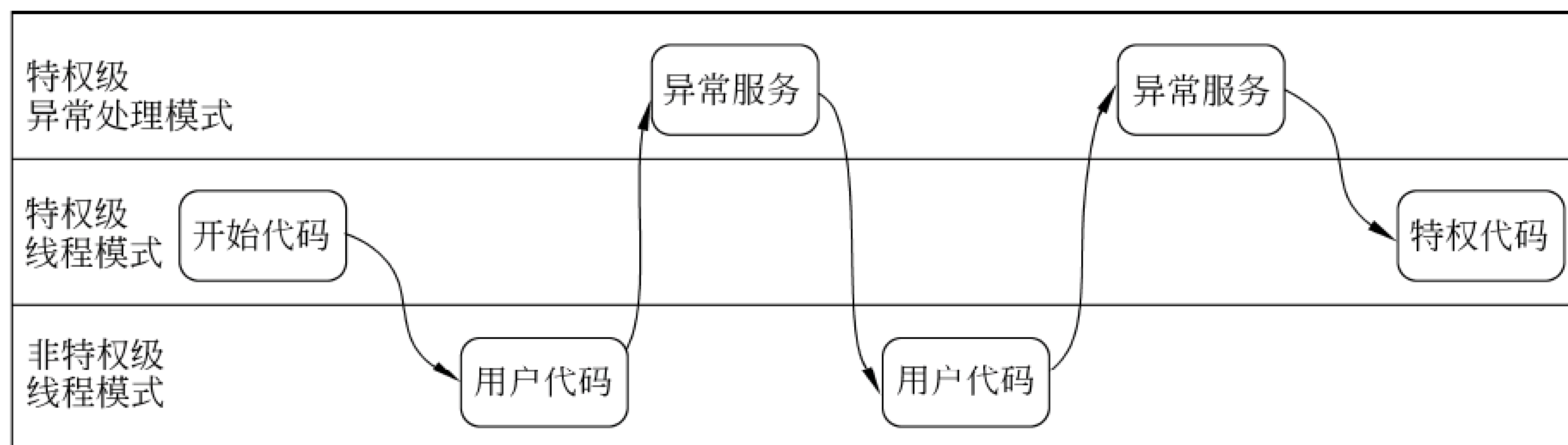


图 2-3 特权级别和非特权级别的转换示例

把代码按特权级和用户级分开，主要是操作系统的需求，操作系统工作在特权级，用户程序工作在非特权级，当用户代码出问题时，不会影响整个系统的运行。结合 MPU，可以防止用户代码访问不属于它的内存区域。

2.2.3 寄存器

Cortex-M3 处理器寄存器堆中寄存器如表 2-3 所示，包括了通用寄存器、特殊功能寄存器以及状态寄存器。不同的寄存器要求在不同特权等级下进行访问，例如，PRIMASK 优先权屏蔽寄存器只能在特权级下才能进行访问。

表 2-3　Cortex-M3 处理器中的寄存器

寄　存　器		访 问 类 型	要求的特权等级	重　置　值
R0～R12		RW	特权级或非特权级	不确定
R13	MSP	RW	特权级	不确定
	PSP	RW	特权级或非特权级	不确定
LR(R14)		RW	特权级或非特权级	0xFFFFFFFF
PC(R15)		RW	特权级或非特权级	不确定
PSR	APSR	RW	特权级或非特权级	不确定
	IPSR	RO	特权级	0x00000000
	EPSR	RO	特权级	0x01000000
PRIMASK		RW	特权级	0x00000000
FAULTMASK		RW	特权级	0x00000000
BASEEPRI		RW	特权级	0x00000000
CONTROL		RW	特权级	0x00000000

如图 2-4 所示，Cortex-M3 的 16 个通用寄存器 R0～R15，分为通用寄存器 R0～R12，特殊寄存器 R13～R15。

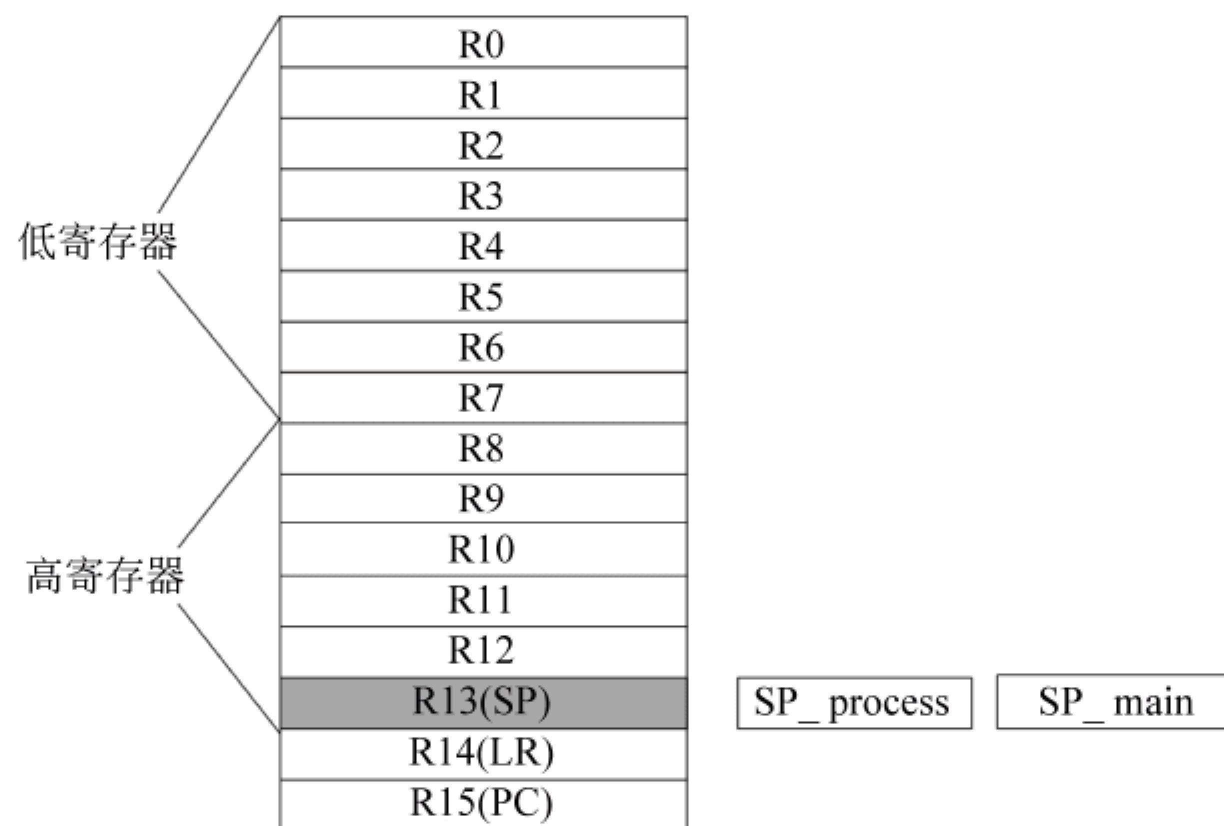


图 2-4　寄存器堆

1. 通用寄存器 R0～R12

通用寄存器 R0～R12 可以被大多数指令使用，按照 16 位和 32 位指令的划分，通用寄存器分为两段：

- 低组寄存器 R0～R7：可以被所有的指令访问。
- 高组寄存器 R8～R12：可以被所有 32 位指令访问，不能被 16 位指令访问。

【思考题：寄存器为何要分为两段?**】**

2. 栈指针寄存器 R13

为了避免操作系统的栈因应用程序的错误使用而毁坏，应用程序和操作系统的栈不共享栈指针，即 R13 寄存器（也记作 SP）在特权模式和用户模式下分别对应不同的栈指针寄存器，这两个栈指针在特权级别变化时自动切换。Cortex-M3 处理器的两个栈指针分别为：

（1）主栈指针 SP_main，记作 MSP（Main Stack Pointer）。这是默认的栈指针，它由 OS 内核、异常服务例程以及所有需要特权访问的应用程序代码来使用。

（2）进程栈指针 SP_process，记作 PSP（Process Stack Pointer），用于应用程序代码。

异常处理模式下始终使用 MSP，而线程模式可配置为 MSP 或 PSP。处理器复位后，处于特权级线程模式，所有代码都使用主栈 MSP。异常处理模式，异常处理程序 ISR 可以通过改变其在退出时使用的 EXC_RETURN 值来指定返回到线程模式时使用的栈。此外，在线程模式中，使用 MSR 指令对控制寄存器的第二位 CONTROL[1]执行写 1 操作也可以从主栈切换到进程栈。因此，栈指针 R13 是分组寄存器，在 SP_main 和 SP_process 之间切换。在任何时候，进程栈和主栈中只有一个是可见的，由 R13 指示。

栈指针寄存器 R13 中存放的是当前栈的地址，PUSH 和 POP 指令会自动对 R13 进行操作，其最低两位 R13[1：0]被强制置零，即它保存的地址是 4 字节对齐的。

3. 链接寄存器 R14

R14 链接寄存器（也记作 LR）用来保存返回地址。当主函数调用一个子函数时，就将主函数的下一条指令的地址（即子程序完成后的返回地址）保存到 LR，当子函数调用结束时返回到该地址便可继续执行主调函数；在发生异常中断时，LR 也用于特殊用途。其他任何时候都可以将 R14 看作一个通用寄存器。

4. 程序计数器 R15

R15 程序计数器（也记作 PC）指向下一条将要被执行的指令地址，该寄存器的最低位始终为 0，因此，指令始终与字或半字边界对齐，即指令是 16 位或 32 位的。如果向 PC 中写入地址，就会引起一次程序的跳转。读取 PC 将返回当前指令地址+4 的值，例如：

```
0x1000: MOV R0, PC
```

则

```
R0 = 0x1004
```

【思考题：读取 PC 将返回当前指令地址+4 的值原因是什么?】

5. 特殊功能寄存器组

Cortex-M3 的特殊功能寄存器包括：程序状态寄存器组 xPSR、中断屏蔽寄存器组（PRIMASK、FAULTMASK 以及 BASEPRI）和控制寄存器（CONTROL），这些寄存器只能通过状态寄存器操作指令 MSR 和 MRS 在特权级别下访问。

（1）xPSR 程序状态寄存器由 3 个独立的状态寄存器组合而成，分别为应用程序状态寄存器（Application Program Status Register，APSR）、中断程序状态寄存器（Interrupt Program Status Register，IPSR）和执行程序状态寄存器（Execution Program Status

Register,EPSR)。其中,27～31 位为 APSR 所用,9～26 位为 EPSR 所用,0～8 位为 IPSR 所用。

APSR 是应用程序状态寄存器,用来保存条件代码标志。APSR 寄存器的有效域定义如图 2-5 所示。

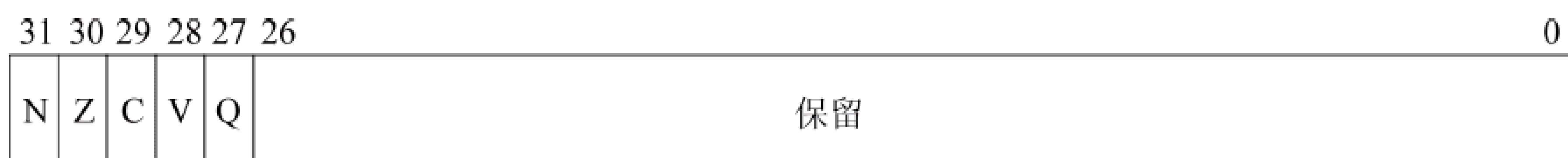


图 2-5　APSR 寄存器定义

N:负数或小于标志,1 表示结果为负数或小于,0 表示结果为整数或大于。

Z:零表示,1 表示结果为 0,0 表示结果非 0。

C:进位标志,1 表示有进位,0 表示没有进位。

V:溢出标志,1 表示有溢出,0 表示没有溢出。

Q:黏着饱和(sticky saturation)标志。

(2) 中断状态寄存器 IPSR,用于存放当前激活的异常的 ISR 编号。IPSR 的有效域定义如图 2-6 所示。

31　9　8　0

保留　中断号

图 2-6　IPSR 寄存器定义

该寄存器的 0～8 位有效,用于表示中断服务编号,该域的值为 0 表示无中断,2 表示非屏蔽中断 NMI,16～255 表示 240 个外部中断。

(3) 执行状态寄存器 EPSR,该寄存器的 24～26 位,10～15 位有效,用于保存多寄存器连续加载过程中产生中断的相关信息。

3 个寄存器分别使用 32 位的不同区域,3 个寄存器可以单独访问,也可以 2 个或 3 个一起访问,在处理器进入异常时,处理器将 3 个状态寄存器组合的信息压入栈进行保存。

(4) 中断优先权屏蔽寄存器 PRIMASK 只有一个有效位,最低位为 1 时,关闭所有可屏蔽中断,只响应不可屏蔽中断 NMI 和硬件错误异常。

(5) 异常屏蔽寄存器 FAULTMASK 只有一个有效位,最低位为 1 时,关闭所有除不可屏蔽中断 NMI 外的所有异常。

(6) 中断优先级基准寄存器 BASEPRI 用来定义优先级的阈值,所有优先级大于该值的中断被关闭(优先级号越大,优先级越低)。

(7) 控制寄存器 CONTROL 只有两位有效,分别用于控制所使用的栈指针以及处理器工作在线程模式时的特权等级。CONTROL 的 bit[0]为 0 表示程序执行在特权级,bit[0]为 1 表示程序执行在非特权级。CONTROL 的 bit[1]为 0,表示选择主栈指针(MSP),bit[1]为 1,表示选择进程栈指针(PSP)。在异常响应模式下,只允许使用 MSP,所以此时不得往该位写 1。

2.2.4 总线

如图 2-7 所示，CM3Core 通过总线矩阵与代码存储器、数据存储器、外设和芯片内部外设的互联。ARM 处理器使用的总线规范是 AMBA，AMBA 规范主要包括 AHB 和 APB 两套总线。

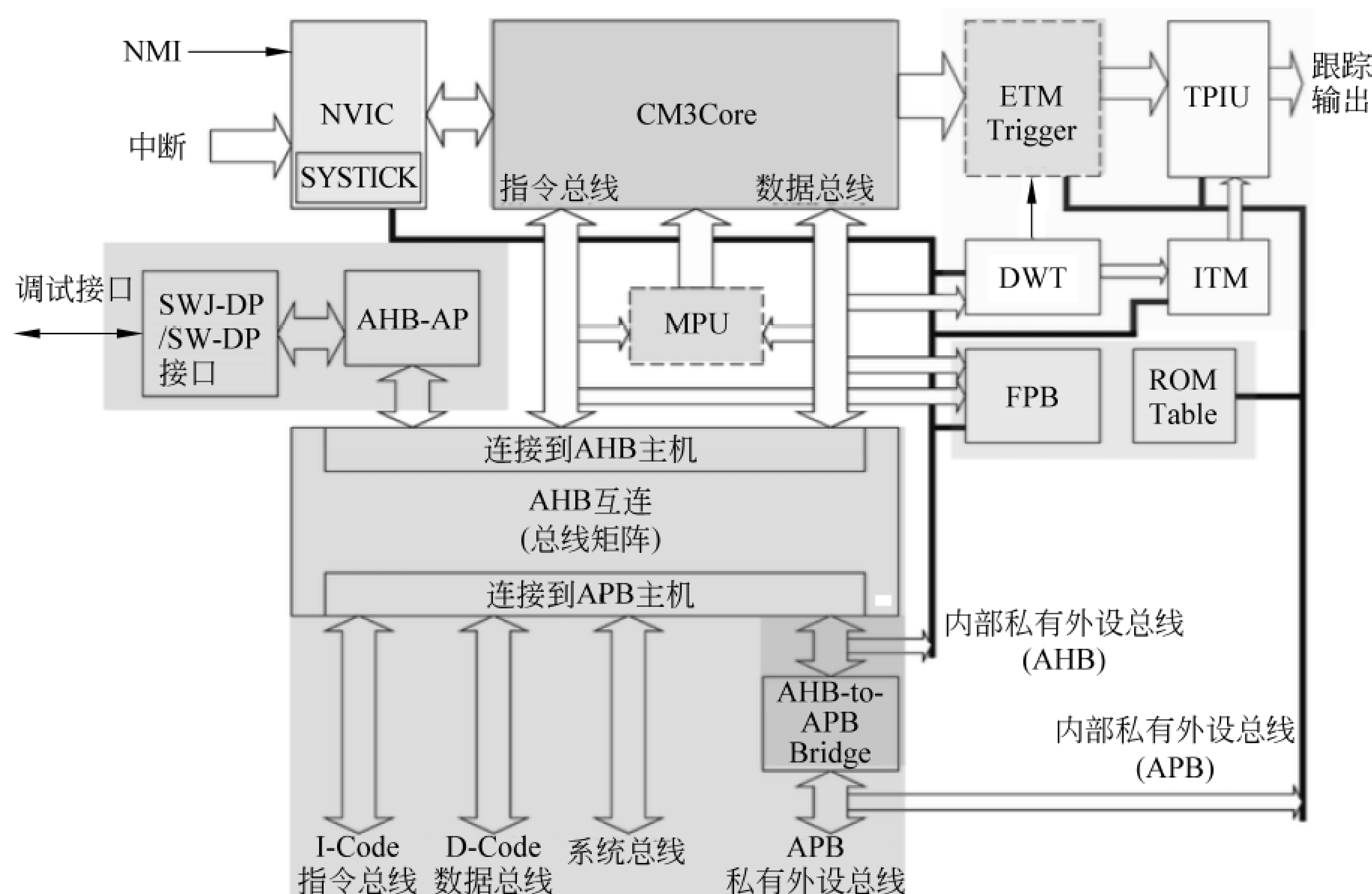


图 2-7　Cortex-M3 内部总线连接

AHB(Advanced High Performance Bus)用于高性能模块(如 CPU、DMA 和 DSP 等)之间的连接，其主要特性有：支持突发传输、分段传输，支持多个主控制器，可配置 32～128 位总线宽度等。AHB 是一套主从控制的总线，整个 AHB 总线上的传输都由主模块发出，由从模块负责回应，多个主从设备之间的数据通信由总线仲裁器进行管理。

AHB-Lite 是 AHB 总线的一个简化版本，只支持一个主模块，且没有总线仲裁器，简化了总线的复杂度。

APB(Advanced Pripheal Bus)总线主要用于低带宽的周边外设之间的连接，例如串口、USB 等，总线控制逻辑简单，只支持一个主模块 AHB/APB 桥接器，AHB 和 APB 通过桥接器进行数据连接和信号转换。

I-Code 总线和 D-Code 总线是 Cortex-M3 哈佛结构取指令和存取数据的两条基于 AHB-Lite 协议的 32 位总线，两条总线的地址访问范围均为 0x00000000～0x1FFFFFFF。I-Code 是指令总线，对于 16 位 Thumb 指令，一次可取两条指令；D-Code 是 32 位数据总线。

系统总线也是一条基于 AHB-Lite 总线协议的 32 位总线，负责在 0x20000000～0xDFFF_FFFF 和 0xE0100000～0xFFFFFFFF 两段地址空间的所有数据传送。

外部私有外设总线 PPB 是一条基于 APB 总线协议的 32 位总线。此总线用于 TPIU、ETM 以及 ROM 表等调试接口及外设。

一个典型的 Cortex-M3 和外部设备的总线连接实例如图 2-8 所示。

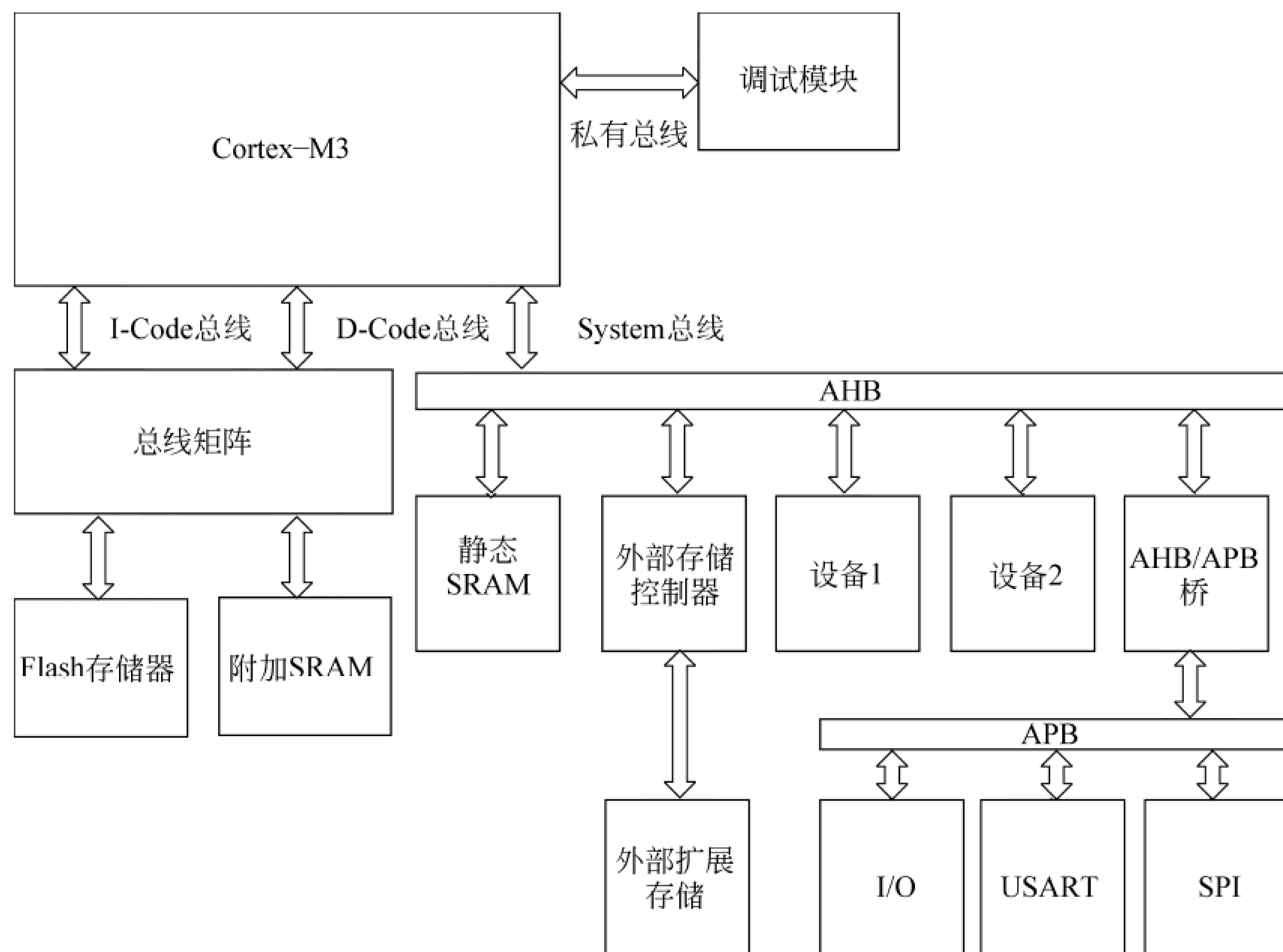


图 2-8　典型的总线连接结构

代码存储器既可以由指令总线(I-Code)访问,也可以被数据总线(D-Code)访问,总线矩阵可以实现指令和数据总线的分离。通过 AHB 总线矩阵把取指和数据访问分开后,如果指令总线和数据总线在同一时刻访问不同的存储器设备(例如,从 Flash 中取指的同时从附加的 SRAM 中访问数据)。但在一些系统实现时,没有附加 SRAM 或者使用了简化的总线复用器,则数据和指令无法同时进行传输。但微控制器内部集成的静态 RAM 一般连接在系统总线上,此时可以用 I-Code 访问 Flash 存储器、用 D-Code 访问系统总线 AHB 的 SRAM,实现哈佛结构。

为了增加系统的存储空间,可以在 AHB 总线上连接一个外部存储控制器,对外提供外部总线连接接口,实现 RAM 或 Flash 的扩容。

【思考题:为何代码存储器既可以由指令总线访问,也可以被数据总线访问?】

2.2.5　存储器

1. 数据对齐

Cortex-M3 支持 32 位的字、16 位半字和 8 位的字节操作。通常情况下,要求总线上的数据要对齐,即以字为单位进行数据传输,其地址的最低两位必须是 0(地址是 4 的整数

倍);以半字为单位的传送,其地址的最低位必须是 0(地址是 2 的倍数)。如果使用了奇数地址,在一些处理器如 ARM7TDMI 中,会产生异常。Cortex-M3 支持非对齐的数据传输,其内部实际是通过把非对齐的访问转换成多个若干对齐的访问实现的,如图 2-9 的非对齐存储,需要通过两次存储器操作才能拿到数据,转换由总线单元完成,对程序员透明,但不是所有的地址空间都可以非对齐访问。一般情况下,我们采用对齐访问的方式,提高 CPU 的执行效率。

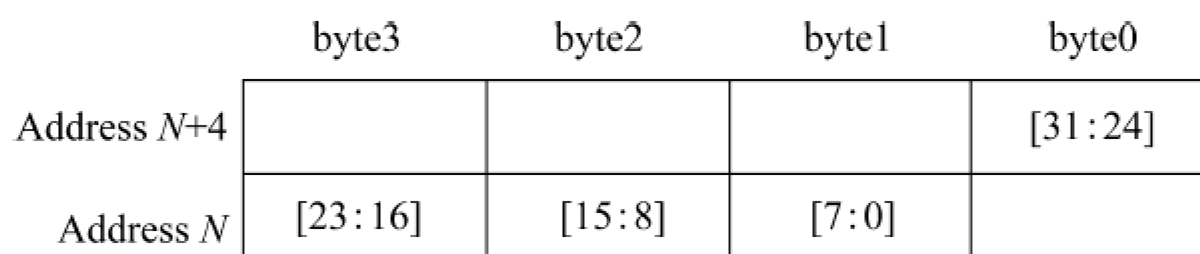


图 2-9 非对齐数据访问

2. 存储器格式

存储器是一个以字节为单位的线性存储空间,其地址可以从 0 开始向上编号,例如,字节 0~3 存放第一个被保存的字,字节 4~7 存放第二个被保存的字。但在存储半字和字的时候,一个半字或字的高字节和低字节的排列顺序可以不同,即存储器的大端和小端存储模式。

(1) 在小端格式中,一个字中最低地址的字节为该字的最低有效字节,最高地址的字节为最高有效字节,如图 2-10 所示。存储器系统地址 0 的字节与数据线 0~7 相连。

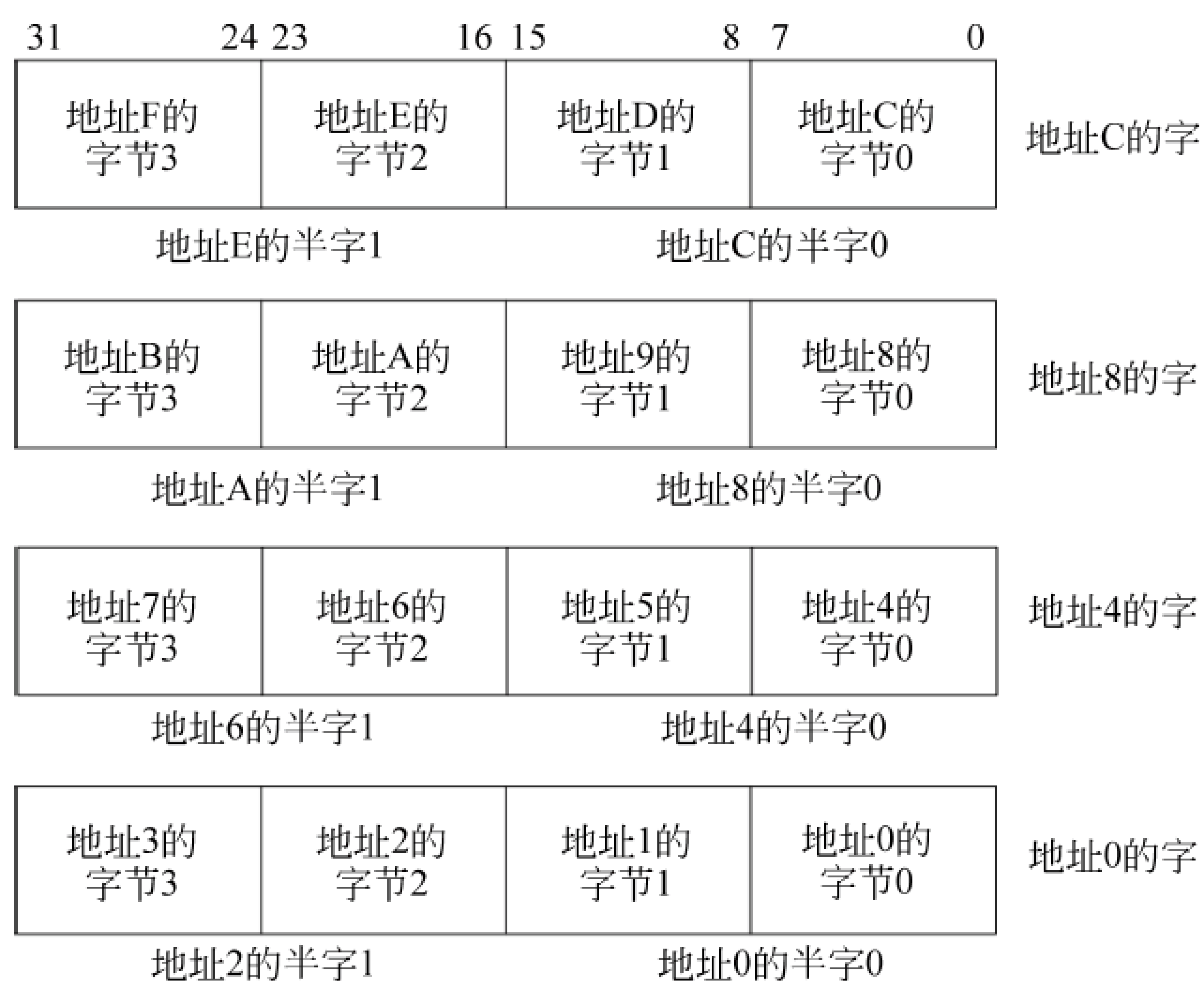


图 2-10 小端存储模式

(2) 在大端格式中,一个字中最低地址的字节为该字的最高有效字节,而最高地址的字节为最低有效字节,如图 2-11 所示。存储器系统地址 0 的字节与数据线 24~31 相连。

Cortex-M3 处理器有一个配置引脚 BIGEND,可以使用它来选择小端格式或大端格式。小端格式是 ARM 处理器默认的存储器格式。Cortex-M3 处理器能够以小端格式或大端格式访

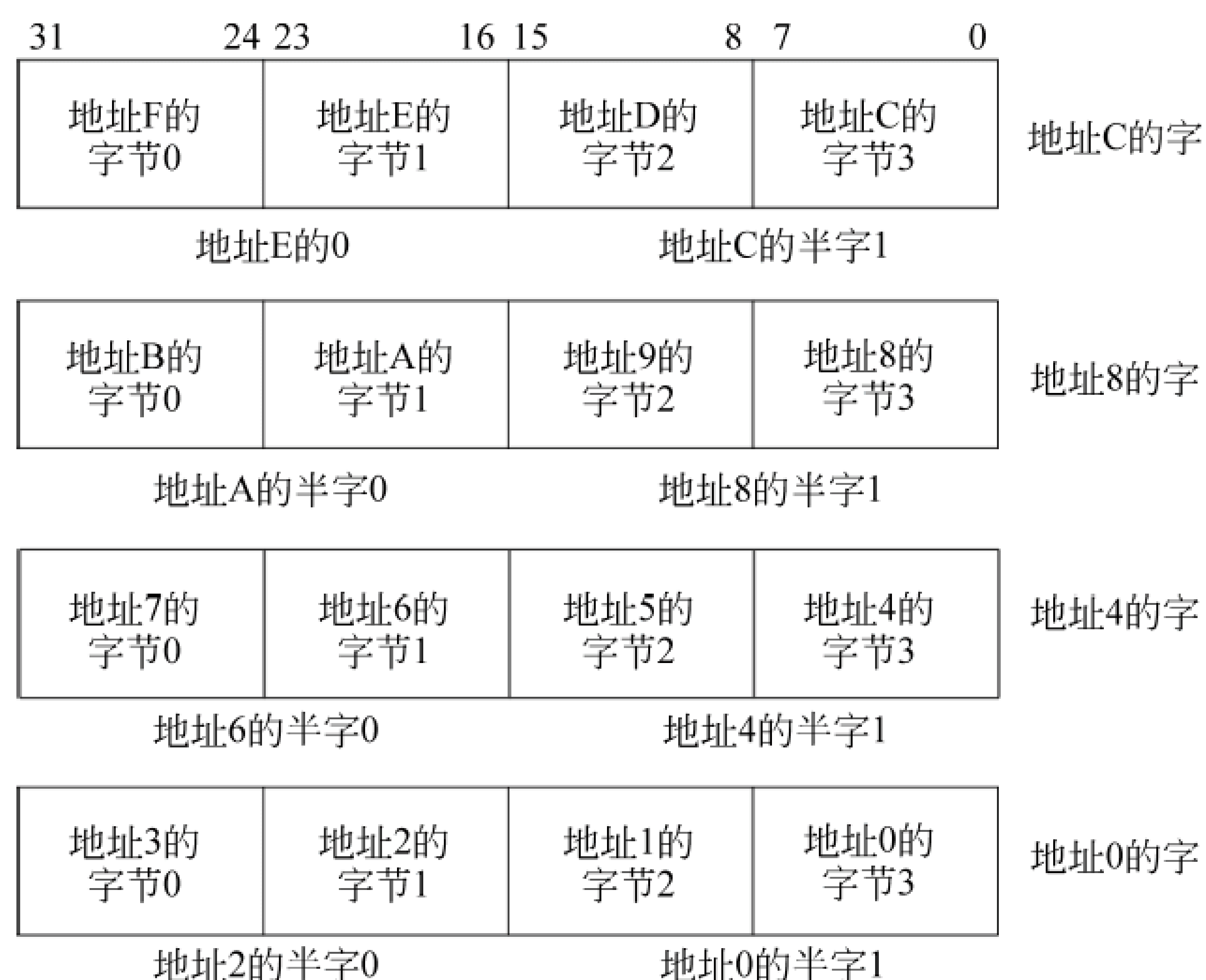


图 2-11　大端存储模式

问存储器中的数据字，而访问代码、系统控制空间 SCS 以及私有外设总线 PPB 空间时必须使用小端格式。

3. 存储器空间分配

Cortex-M3 采用统一编址，并对存储器空间分配进行了规范，Flash、SRAM 等起始地址以及 NVIC、MPU 的外设地址规定了具体的起始范围，这样使得不同厂家微控制器芯片的存储映射大体相同，便于在不同厂家微控制器之间的程序移植。

图 2-12 为 Cortex-M3 的存储器映射。

表 2-4 列出了被不同的存储器映射区域寻址的处理器接口。

表 2-4　存储器接口

存储器映射	接　　口
代码	指令取指在 I-Code 总线上执行，数据访问在 D-Code 总线上执行
SRAM	指令取指和数据访问都在系统总线上执行
SRAM_bitband	SRAM 的别名区域，数据访问是别名，指令访问不是别名
外设	指令取指和数据访问都在系统总线上执行
外设_bitband	外设别名区域，数据访问是别名，指令访问不是别名
外部 RAM	指令取指和数据访问都在系统总线上执行
外部设备	指令取指和数据访问都在系统总线上执行
专用外设总线	对 ITM、NVIC、FPB、DWT、MPU 的访问在处理器内部专用外设总线上执行。对 TPIU、ETM 和 PPB 存储器映射的系统区域的访问在外部专用外设总线上执行
系统	厂商系统外设的系统部分

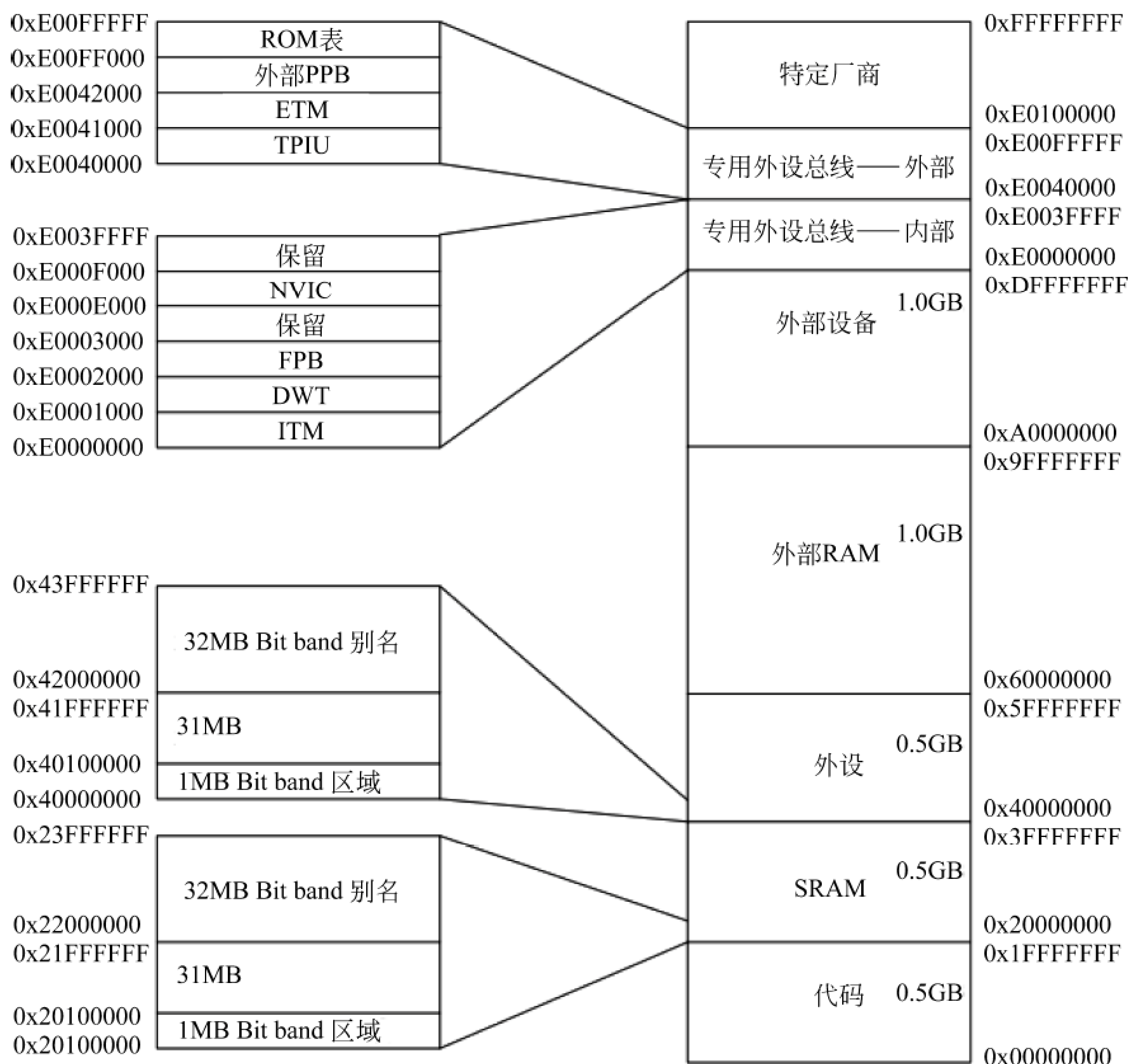


图 2-12　Cortex-M3 存储器映射

4. bit-banding 位带操作

bit-banding 技术是一种实现数据直接进行位操作的加速技术。如图 2-13 所示，存储器

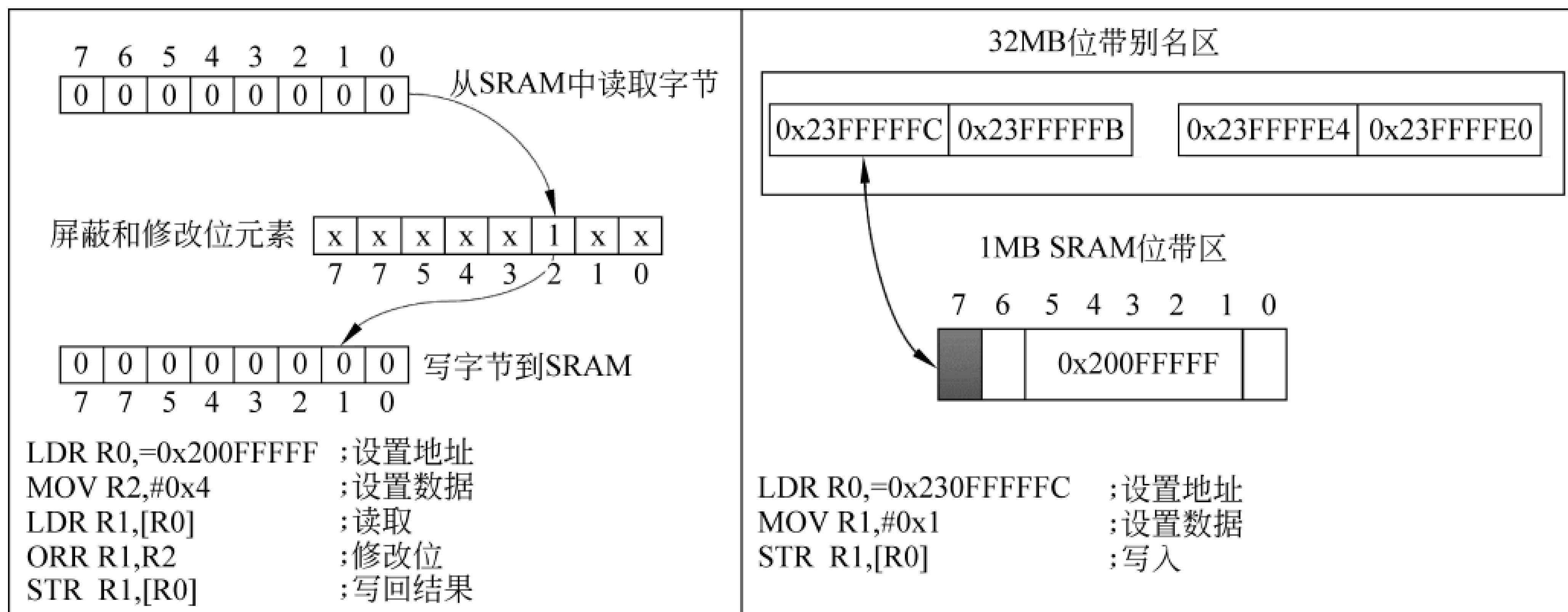


图 2-13　bit-banding 的对比

的最小访问单位是字节，通常情况下，要对一个字节中的某个 bit 进行操作，需要先读出该字节，对其中的某些位进行修改后再将整个字节写入到存储器中。Cortex-M3 的位带是在一段存储空间 A 中，将一个字的每一位(bit)映射到另一个存储空间 B 中的一个字(32bit)，对于 B 中字的操作即是对 A 中字的某一位的操作，从而通过一次读写即可实现对字节内部 bit 的直接操作。位带技术对于外设控制器的寄存器控制是非常有用的。我们把 B 称为位带别名区域。

Cortex-M3 存储器映射有 2 个 32MB 别名区，它们分别对应两个 1MB 的 bit-band 区。对 32MB SRAM 别名区的访问映射为对 1MB SRAM bit-band 区的访问。对 32MB 外设别名区的访问映射为对 1MB 外设 bit-band 区的访问。别名区中的字与 bit-band 区中对应的位的映射公式如下：

```
bit_word_offset= (byte_offset× 32)+ (bit_number× 4)
bit_word_addr=bit_band_base+bit_word_offset
```

其中，

- bit_word_offset 为 bit-band 存储区中的目标位的位置；
- bit_word_addr 为别名存储区中映射为目标位的字的地址；
- bit_band_base 是别名区的开始地址；
- byte_offset 为 bit-band 区中包含目标位的字节的编号；
- bit_number 为目标位的位位置(0～7)。

图 2-14 显示了 SRAM 位带别名区和 SRAM 位带区之间映射的一个例子：

别名区地址 0x23FFFFE0 的字映射为 0x200FFFFC 的 bit-band 字节的位 0：0x23FFFFE0＝0x22000000＋(0xFFFF＊32)＋0＊4；别名区地址 0x23FFFFEC 的字映射为 0x200FFFFC 的 bit-band 字节的位 7：0x23FFFFFC＝0x22000000＋(0xFFFF＊32)＋7＊4。

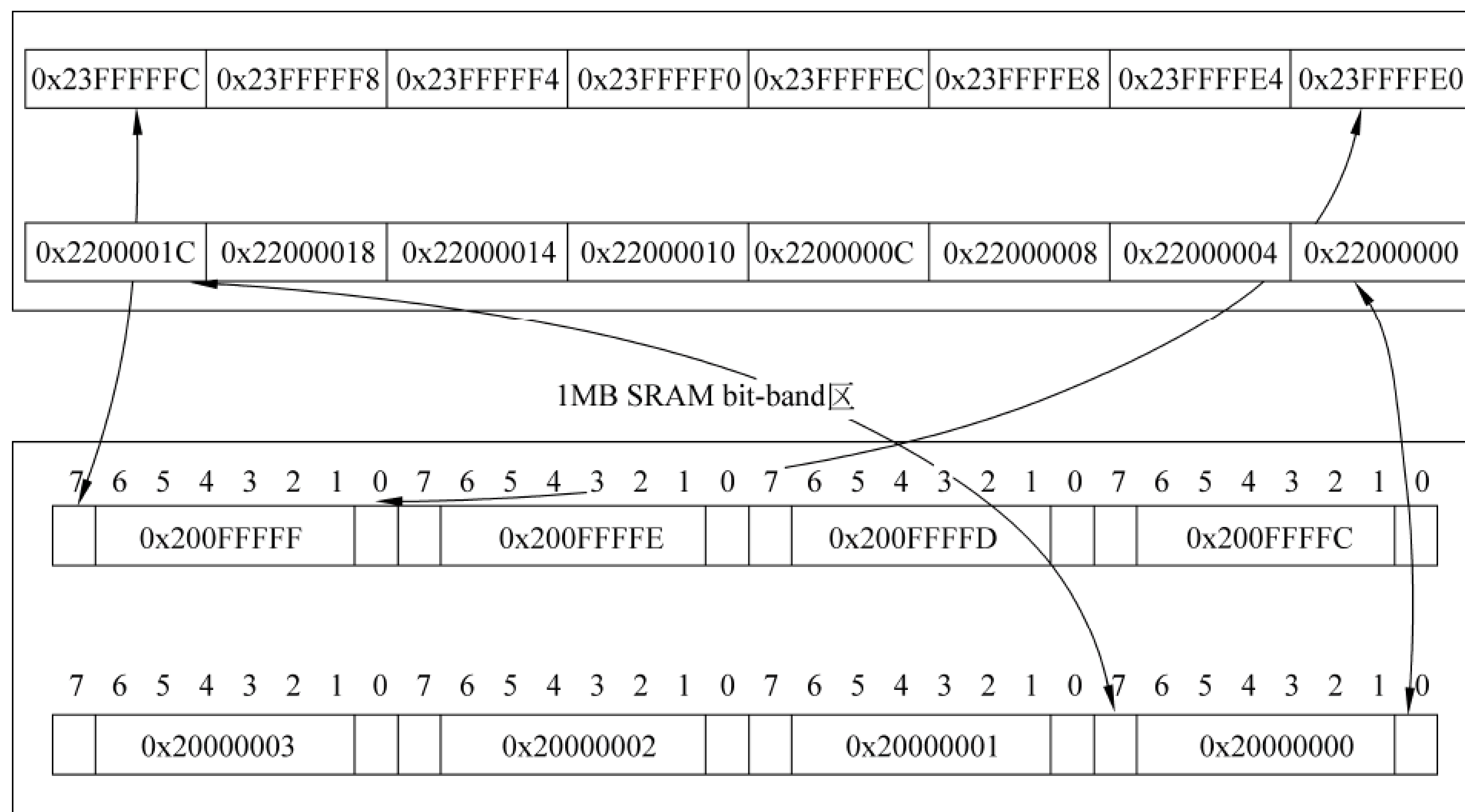


图 2-14　位带映射示例

2.2.6 中断

Cortex-M3处理器使用一个可以重定位的向量表，表中包含了将要执行的中断服务程序的函数地址。中断被响应后，处理器通过指令总线接口从向量表中获取地址。向量表复位时存储地址为零，通过配置特殊寄存器可以使向量表重新定位。

当异常发生时，程序计数器、状态寄存器、链接寄存器和R0～R3、R12等通用寄存器将被压进栈。在数据总线对寄存器压栈的同时，处理器识别并定位异常向量，获取中断服务程序代码的第一条指令。一旦压栈和取指完成，中断服务程序就开始执行，中断服务程序执行完成，压栈的寄存器自动出栈恢复，中断了的程序也因此恢复正常的执行。

NVIC支持中断嵌套(通过压栈实现)，允许通过提高中断的优先级对中断进行提前处理。它还支持中断的动态优先权重置。优先权级别可以在运行期间通过软件进行修改。在两个同级别中断发生的情况中，传统的系统将重复状态保存和状态恢复的过程两次，导致了延迟的增加。Cortex-M3处理器使用末尾连锁(tail-chaining)技术简化了正在实行和将要执行的中断之间的移动。末尾连锁技术把需要用时30个时钟周期才能完成的连续的栈弹出和压入操作替换为6个周期就能完成的指令取指，实现了延迟的降低。处理器状态在进入中断时自动保存，在中断退出时自动恢复，比软件执行用时更少，大大提高了系统的性能，NVIC中断控制器的细节将在第6章中断中进行详细介绍。

2.3 STM32L152RET6微处理器介绍

本教材选用了意法半导体公司的STM32L系列超低功耗处理器STM32L152RET6，该处理器属于Cortex-M3系列。

对于超低功耗处理器STM32L152RET6，其涉及的命名规则如下：

(1) STM32表示基于ARM的32位微处理器。

(2) L表示低功耗产品系列，F表示通用产品系列。

(3) 152是产品系列中子类型产品。

后续的四位编号RET6的说明如下：

(4) 第一位表示引脚数量：R表示64引脚，T表示36引脚，C表示46引脚，V表示100引脚，Z表示144引脚。

(5) 第二位表示Flash大小：B表示Flash容量为128KB，6表示Flash容量为32KB，8表示Flash存储容量为64KB，C表示Flash存储容量为256KB，D Flash存储容量384KB，E表示Flash存储容量为512KB，G表示Flash存储容量为1MB。

(6) 第三位表示封装方式：T表示LQFP封装，H表示BGA封装，U表示VFQFPN封装。

(7) 第四位表示工作温度：6表示工业级，工作温度为－40℃～85℃；当为7时，表示工

作温度为－40℃～105℃。

STM32L152RET6 主要配置包括：512KB 的 Flash、80KB 的 RAM、16KB EEPROM、2 个超低功耗比较器、6 个通用计时器和 2 个基本计时器、2 个 SPI 通信接口、2 个 I2C 通信接口、3 个 USART 通信接口和 1 个 USB 接口、51 个常规输入输出端口(分为 6 组)、1 个 12 位 20 通道的模数转换器、2 个 12 位 2 通道的数模转换器、主频最大为 32MHz、工作电压在 1.8～3.6V 之间。

当然，作为超低功耗处理器 STM32L152RET6，其支持 7 种类低功耗工作模式，待机模式最低功耗可达 290nA，唤醒时间 8uS：

(1) 睡眠模式：只有 CPU 停止工作，所有其他外部设备继续运行，当中断或事件发生时可以唤醒 CPU。

(2) 低功耗运行模式：内部时钟工作频率为 65kHz，外部使能设备数量受限。

(3) 低功耗睡眠模式：在睡眠模式下通过将内部电压调节器设置为低功耗模式以减少电压调节器的工作电流，外部使能设备数量受限。

(4) 带 RTC 的停止模式：同时保持 RAM 和寄存器的内容以及实时时钟，低速时钟工作，电压调节器工作在低功耗模式可以由外部中断唤醒。

(5) 不带 RTC 的停止模式：保持 RAM 和寄存器的内容，所有时钟均停止工作，电压调节器工作在低功耗模式，可以由外部中断唤醒。

(6) 带 RTC 的待机模式：内部电压调节器被关闭。低速时钟仍然运行，RAM 和大多数寄存器的内容丢失；待机模式可由复位(NRST 引脚信号、独立看门狗)、唤醒引脚以及 RTC 事件触发退出。

(7) 不带 RTC 的待机模式：内部电压调节器被关闭，所有时钟均停止工作，RAM 和大多数寄存器的内容会丢失。这种待机模式可由外部复位(NRST 引脚信号)或唤醒引脚来触发退出。

2.4　STM32L152RET6 微处理器的系统结构

STM32L152RET6 微处理器的系统结构，如图 2-15 所示。

I-Code 总线将 Cortex-M3 内核与闪存(Flash)指令接口相连接，指令通过该总线传输。D-Code 总线将 Cortex-M3 内核与 Flash 数据接口相连接。System 总线连接 Cortex-M3 内核的 System 总线到总线矩阵，总线矩阵协调 Cortex-M3 内核和 DMA 的访问。DMA 总线将连接 DMA 到总线矩阵。

总线矩阵协调 Cortex-M3 的 System 总线和 DMA 总线之间的访问仲裁。仲裁采用轮换算法。总线矩阵包含五个主动部件，包括 Cortex-M3 的 D-Code 总线、Cortex-M3 的 System 总线、以太网 DMA 总线、DMA1 总线和 DMA2 总线，三个从属部件，包括 Flash 接口、SRAM 和 AHB/APB 桥。

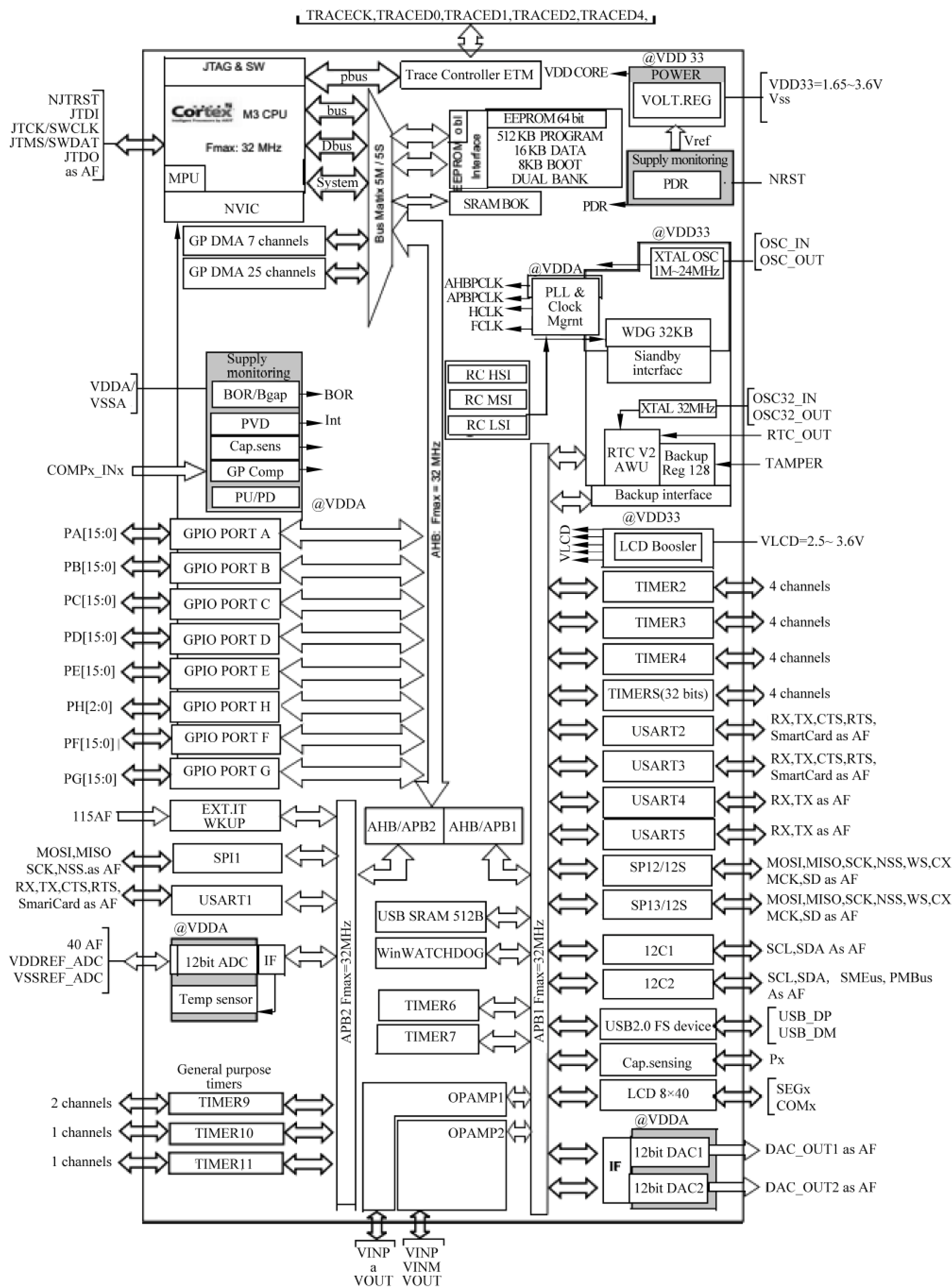


图 2-15 STM32L152RET6 的系统结构

两个AHB/APB桥在AHB和2个APB总线间提供同步连接。当APB进行8位或16位访问时，该访问会自动转换成32位的访问。

APB总线上连接的外设主要有：数模转换器(Digital Analog Converter，DAC)、电源(Power，PWR)、USB(Universal Serial Bus)设备、I2C(Inter Integrate Circuit)总线、USART(Universal Serial Asynchronous Receiver Transmitter)总线、SPI(Serial Peripheral Interface)总线、IWDG(Independent Watchdog)独立看门狗、WWDG(Window Watchdog)窗口看门狗、RTC(Real Time Clock)实时时钟、实时器(Timer)等。

I2C总线是由PHILIPS公司开发的两线式串行总线，一条是SDA(Serial Data)串行数据线，另一条是SCL(Serial Clock)串行时钟线，用于连接微控制器及其外围设备。I2C是微电子通信控制领域广泛采用的一种总线标准，是同步通信的一种特殊形式，具有接口线少，控制方式简单，器件封装形式小，通信速率较高等优点。

USART是一种通用串行数据总线，用于异步通信。该总线双向通信，可以实现全双工传输和接收。在嵌入式设计中，USART用来与PC进行通信，包括与监控调试器和其他器件，如EEPROM通信。USART首先将接收到的并行数据转换成串行数据来传输。消息帧从一个低位起始位开始，后面是7个或8个数据位，一个可用的奇偶位和一个或几个高位停止位。接收器发现开始位时它就知道数据准备发送，并尝试与发送器时钟频率同步。如果选择了奇偶，UART就在数据位后面加上奇偶位。奇偶位可用来帮助错误校验。接收过程中，UART从消息帧中去掉起始位和结束位，对进来的字节进行奇偶校验，并将数据字节从串行转换成并行。UART也产生额外的信号来指示发送和接收的状态。例如，如果产生一个奇偶错误，UART就置位奇偶标志。

SPI总线是一种高速的、全双工、同步的通信总线，并且在芯片的引脚上只占用四根线，节约了芯片的引脚，同时为PCB的布局上节省空间，提供方便。

独立看门狗IWDG是一个12位的向下计数器，其时钟来自于其内部独立的37kHz的RC振荡器(只用电阻和电容构成的振荡器称为RC振荡器)。该看门狗可用于当故障发生时重置设备，还可用于应用超时管理的自激时钟。独立看门狗没有中断。一般主要用于监视硬件错误。

窗口看门狗WWDG是一个7位的向下计数器，可用于当问题发生时重置设备。其时钟来源于主时钟。窗口看门狗有中断。一般主要用于监视软件错误。

STM32的实时时钟(RTC)是一个独立的定时器。STM32的RTC模块拥有一组连续计数的计数器，在相应软件配置下，可提供时钟日历的功能。修改计数器的值可以重新设置系统当前的时间和日期。RTC模块和时钟配置系统(RCC_BDCR寄存器)是在后备区域，即在系统复位或从待机模式唤醒后RTC的设置和时间维持不变。但是在系统复位后，会自动禁止访问后备寄存器和RTC，以防止对后备区域(BKP)的意外写操作。所以在要设置时间之前，先要取消备份区域(BKP)写保护。

2.5　STM32L152RET6 微处理器的引脚说明

STM32L152RET6 微处理器的引脚，如图 2-16 所示。

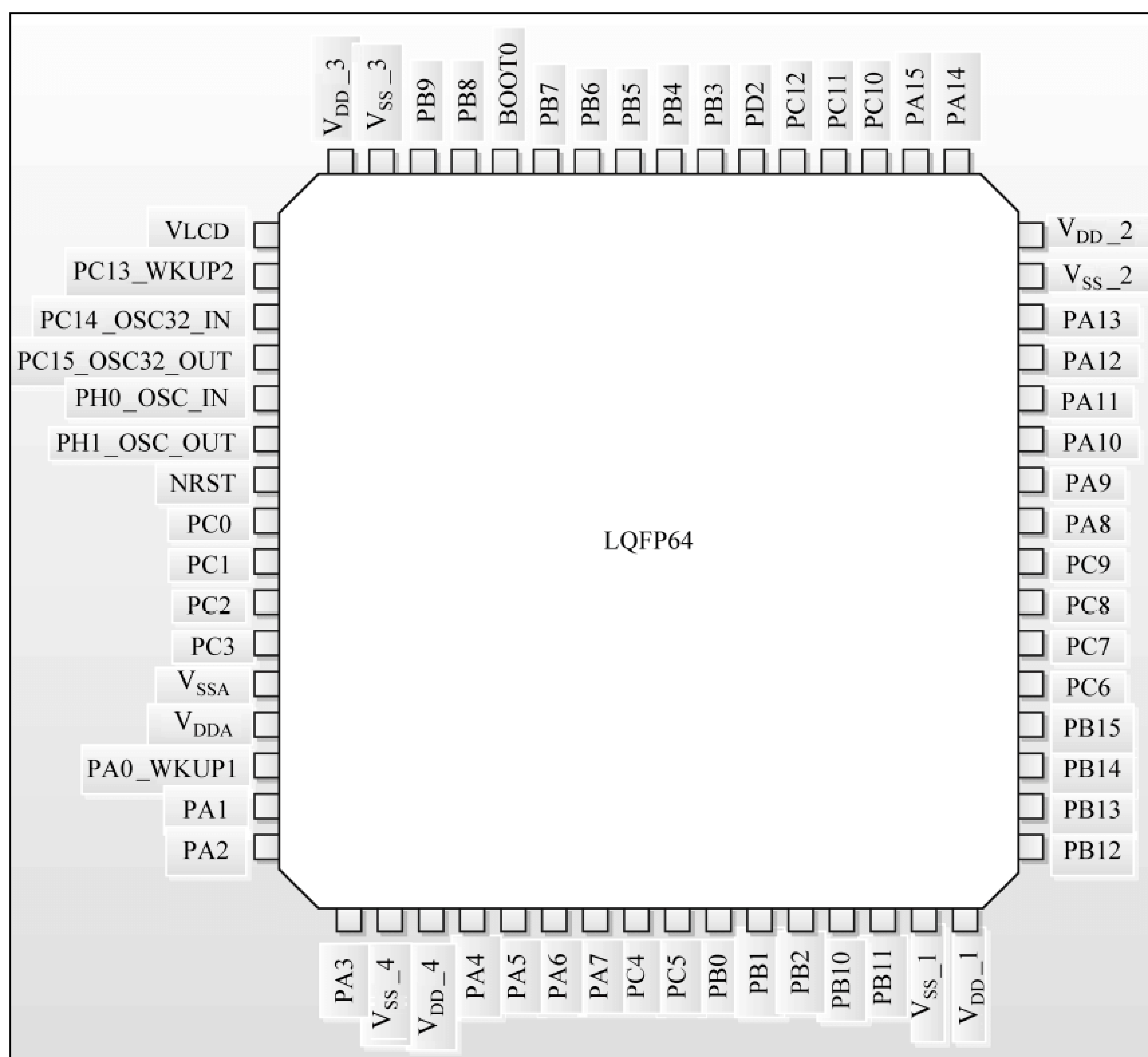


图 2-16　STM32L152RET6 的引脚

各个引脚的具体功能，如表 2-5 所示。其中，引脚类型为 S 表示供电管教，I/O 表示输入输出，I 表示只能输入。I/O 电平 FT 表示可支持 5V 电压，TC 表示支持 3V 电压。

表 2-5　STM32L152RET6 引脚的功能说明

编号	引脚名	引脚类型	I/O 电平	主要功能	附 加 功 能
1	V_{LCD}	S		V_{LCD}	
2	PC13_WKUP2	I/O	FT	PC13	RTC_TAMP1、RTC_TS、RTC_OUT、WKUP2
3	PC14_OSC32_IN	I/O	TC	PC14	OSC32_IN
4	PC15_OSC32_OUT	I/O	TC	PC15	OSC32_OUT

续表

编号	引脚名	引脚类型	I/O 电平	主要功能	附 加 功 能
5	PH0_OSC_IN	I/O	TC	PH0	OSC_IN
6	PH1_OSC_OUT	I/O	TC	PH1	OSC_OUT
7	NRST	I/O	RST	NRST	
8	PC0	I/O	FT	PC0	ADC_IN10、COMP1_INP
9	PC1	I/O	FT	PC1	ADC_IN11、COMP1_INP
10	PC2	I/O	FT	PC2	ADC_IN12、COMP1_INP
11	PC3	I/O	TC	PC3	ADC_IN13、COMP1_INP
12	V_{SSA}	S		V_{SSA}	
13	V_{DDA}	S		V_{DDA}	
14	PA0_WKUP1	I/O	FT	PA0	WKUP1、ADC_IN0、COMP1_INP
15	PA1	I/O	FT	PA1	ADC_IN1、COMP1_INP
16	PA2	I/O	FT	PA2	ADC_IN2、COMP1_INP
17	PA3	I/O	FT	PA3	ADC_IN3、COMP1_INP
18	V_{SS_4}	S		V_{SS_4}	
19	V_{DD_4}	S		V_{DD_4}	
20	PA4	I/O	TC	PA4	ADC_IN4、DAC_OUT1、COMP1_INP
21	PA5	I/O	TC	PA5	ADC_IN5、DAC_OUT2、COMP1_INP
22	PA6	I/O	FT	PA6	ADC_IN6、COMP1_INP
23	PA7	I/O	FT	PA7	ADC_IN7、COMP1_INP
24	PC4	I/O	FT	PC4	ADC_IN14、COMP1_INP
25	PC5	I/O	FT	PC5	ADC_IN15、COMP1_INP
26	PB0	I/O	TC	PB0	ADC_IN8、COMP1_INP、VREF_OUT
27	PB1	I/O	FT	PB1	ADC_IN9、COMP1_INP、VREF_OUT
28	PB2	I/O	FT	PB2/BOOT1	
29	PB10	I/O	FT	PB10	
30	PB11	I/O	FT	PB11	
31	V_{SS_1}	S		V_{SS_1}	

续表

编号	引脚名	引脚类型	I/O 电平	主要功能	附 加 功 能
32	V_{DD_1}	S		V_{DD_1}	
33	PB12	I/O	FT	PB12	ADC_IN18、COMP1_INP
34	PB13	I/O	FT	PB13	ADC_IN19、COMP1_INP
35	PB14	I/O	FT	PB14	ADC_IN20、COMP1_INP
36	PB15	I/O	FT	PB15	ADC_IN21、COMP1_INP、RTC_REFIN
37	PC6	I/O	FT	PC6	
38	PC7	I/O	FT	PC7	
39	PC8	I/O	FT	PC8	
40	PC9	I/O	FT	PC9	
41	PA8	I/O	FT	PA8	
42	PA9	I/O	FT	PA9	
43	PA10	I/O	FT	PA10	
44	PA11	I/O	FT	PA11	USB_DM
45	PA12	I/O	FT	PA12	USB_DP
46	PA13	I/O	FT	JTMS_SWDIO	
47	V_{SS_2}	S		V_{SS_2}	
48	V_{DD_2}	S		V_{DD_2}	
49	PA14	I/O	FT	JTCK_SWCLK	
50	PA15	I/O	FT	JTD1	
51	PC10	I/O	FT	PC10	
52	PC11	I/O	FT	PC11	
53	PC12	I/O	FT	PC12	
54	PD2	I/O	FT	PD2	
55	PB3	I/O	FT	JTDO	
56	PB4	I/O	FT	NJRST	
57	PB5	I/O	FT	PB5	
58	PB6	I/O	FT	PB6	
59	PB7	I/O	FT	PB7	
60	BOOT0	I	B	BOOT0	

续表

编号	引脚名	引脚类型	I/O 电平	主要功能	附 加 功 能
61	PB8	I/O	FT	PB8	
62	PB9	I/O	FT	PB9	
63	V_{SS_3}	S		V_{SS_3}	
64	V_{DD_3}	S		V_{DD_3}	

2.6　STM32L152RET6 微处理器的复位和时钟控制

STM32L152RET6 微处理器支持三种形式的复位：电源复位、系统复位和备份区域复位。

电源复位是指复位所有寄存器。当发生系统上电、掉电或欠压复位时，发生电源复位，这些复位源都作用在 NRST 引脚，提供给设备的系统复位信号都由 NRST 引脚输出。

系统复位是指设置除时钟控制寄存器和备份区域寄存器外的所有寄存器。系统复位可由如下几种方式产生：①引脚 NRST 低电平；②窗口看门狗计数终止；③独立看门狗计数终止；④软件复位；⑤低功耗管理复位，即通过进入待机模式（Standby）或停止模式（STOP）产生的复位；⑥待机模式退出复位。

备份区域复位仅仅设置备份区域寄存器。有以下两种方式产生：①通过设置备份区域控制寄存器 RCC_BDCR 的 BDRST 位置为 1 产生的软件复位；②电源复位。

STM32L152RET6 微处理器共有 5 个时钟，3 个为内部时钟，分别为高速内部时钟（High Speed Internal clock signal，HSI）、低速内部时钟（Low Speed Internal clock signal，LSI）和多速内部时钟（Multi-Speed Internal clock signal，MSI）。两个为外部时钟，分别为高速外部时钟（High Speed External clock signal，HSE）和低速外部时钟（Low Speed External clock signal，LSE）。STM32L152RET6 微处理器的时钟树，如图 2-17 所示。其中，高速外部时钟 HSE 的频率一般在 1M～24MHz 范围内，低速外部时钟 LSE 的频率一般在 0～1000kHz 范围内，一般取典型值 32.768kHz。

【思考题：为何采用多时钟?】

三种不同的主时钟源可被用来驱动系统时钟（SYSCLK）：

（1）16MHz 的 HSI 内部高速振荡器时钟。

（2）1M～24MHz 的 HSE 外部高速振荡器时钟。

（3）MSI 多速率时钟（7 种可配置时钟速率，65.5kHz、131kHz、262kHz、524kHz、1.05MHz、2.1MHz 和 4.2MHz）。

HSE 和 HSI 可以作为 PLL 锁相环的输入源。PLL 是一个倍频模块，可以把低速率时钟倍频后输出高速率时钟，PLL 锁相环的输出时钟经过分频后可以作为 SYSCLK 的时钟源。

【思考题：什么是 PLL？它的原理是什么？**】**

两个辅助时钟源可用于驱动 LCD 控制器以及 RTC 等：

(1) 32.768kHz 的 LSE 时钟。

(2) 37kHz 的 LSI 时钟。

当不使用时，任一个时钟源都可被独立地启动或关闭，由此优化系统功耗。

图 2-17 是 STM32L1xx 系列处理器的时钟树，用户可通过多个预分频器配置 AHB、高速 APB(APB2)和低速 APB(APB1)域的频率。AHB、APB1 和 APB2 域的最大频率是 32MHz，具体取决于芯片的工作电压。所有的外围设备的时钟都由 SYSCLK 主时钟产生，除了以下几种情况：

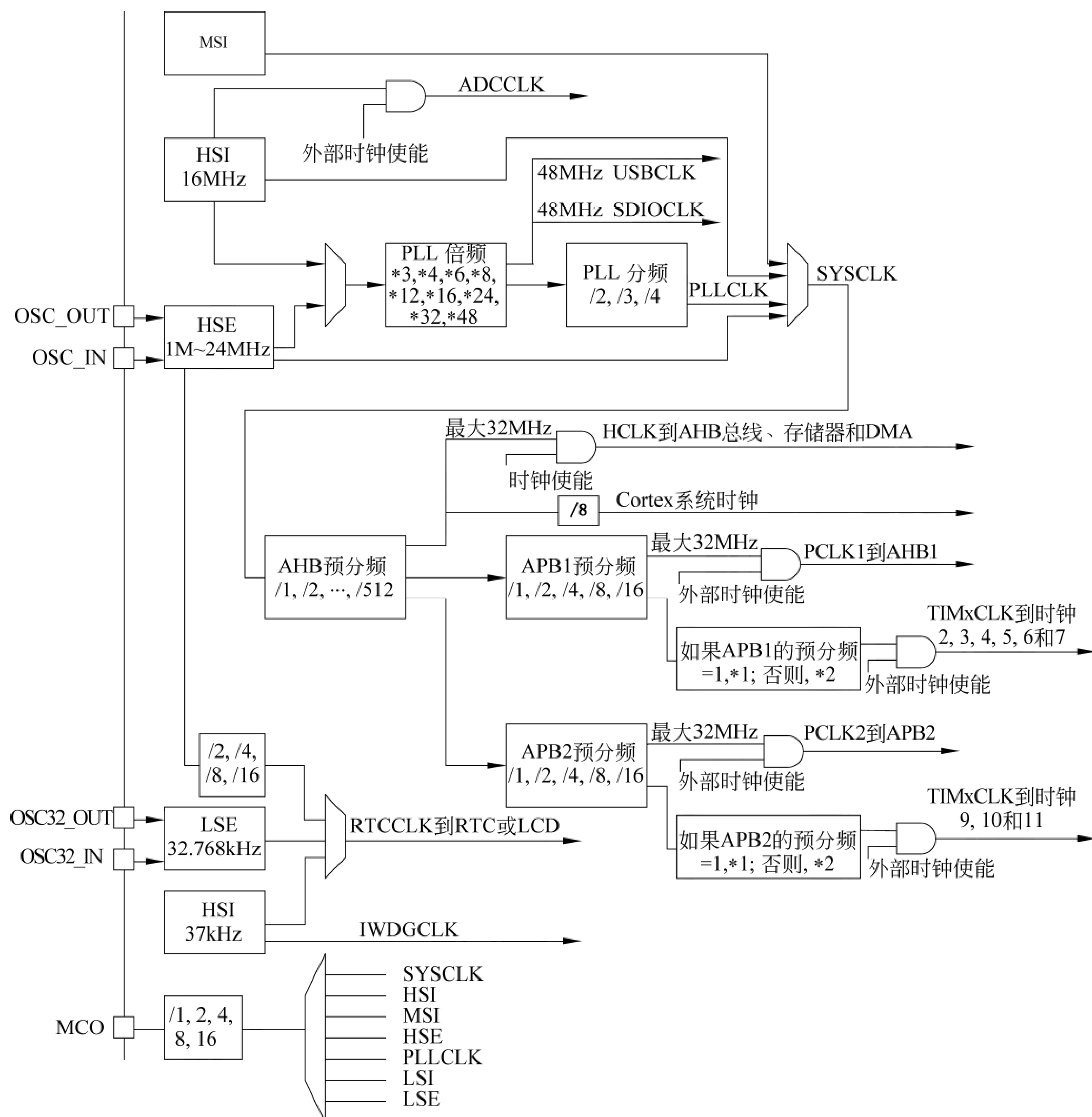


图 2-17 STM32L152RET6 微处理器的时钟树

(1) USB 和 SDIO 的 48MHz 时钟由 PLLVCO 产生。

(2) ADC 由 HIS 时钟 1,2,4 分频产生。

(3) IWDG 由 LSI 时钟提供。

(4) RTC 和 LCD 可以由 LSE、LSI 和 HSE 提供。

时钟由 RCC(Reset and Clock Contorller)控制器进行配置和管理,RCC 的详细介绍见第四章。

2.7 STM32L152RET6 微处理器的存储映射

STM32L152RET6 微处理器的存储映射,如图 2-18 所示。

程序存储器(Flash)、数据存储器(SRAM)、寄存器和输入输出端口(GPIO)统一组织在一个 4GB 的存储区域,即统一编址。如果要访问输入输出端口,就向对应的地址写入数据;如果要设置输入输出端口的属性,就要写信息到相应的寄存器。

代码区始终从地址 0x00000000 开始,且通过 I-Code 和 D-Code 总线访问,而数据区始终从地址 0x2000 0000 开始,且通过系统总线访问。

从地址 0x00000000 到地址 0x1FFFFFFF 共 512MB 空间为块 0,根据 BOOT[1:0]引脚的设置从主闪存存储器(Flash Memory)、系统存储器(System Memory)或内置 SRAM 启动。即 BOOT 的设置将 Flash、System Memory 和 SRAM 映射到 0x00000000 开始的空间。

通过 BOOT[1:0]引脚的设置可以选择三种不同的启动模式,如表 2-6 所示。

表 2-6 STM32L152RET6 的启动模式

BOOT[1:0]引脚设置		启 动 模 式	说　　明
BOOT1	BOOT0		
x	0	主闪存存储器(Flash)	主闪存被选为启动区域
0	1	系统存储器(System Memory)	系统存储器被选为启动区域
1	1	内置 SRAM	内置 SRAM 被选为启动区域

对于不同的启动模式,开始访问的地址空间不同。

(1) 从主闪存存储器(Flash Memory)启动。主闪存存储器被映射到启动空间地址 0x0000 0000～0x07FF FFFF 中,但仍然能够访问原有的从地址 0x0800 0000 开始的空间;

(2) 从系统存储器(System Memory)启动。系统存储器被映射到启动空间地址 0x0000 0000～0x07FF FFFF 中,但仍然能够访问原有的从地址 0x1FF0 0000 开始的空间;

(3) 从内置 SRAM 启动。只能在地址 0x2000 0000 开始的空间访问 SRAM。

从地址 0x2000 0000 到地址 0x3FFF FFFF 共 512MB 空间为块 1,为 SRAM 区。



图 2-18 STM32L152RET6 的存储器映射

从地址 0x4000 0000 到地址 0x5FFF FFFF 共 512MB 空间为块 2，为外部设备区，相关的寄存器组及地址，如表 2-7 所示。

表 2-7　寄存器组地址

地　址	外　设	总线
0x40000000～0x400003FF	TIM2 定时器	APB1
0x40000400～0x400007FF	TIM3 定时器	APB1
0x40000800～0x40000BFF	TIM4 定时器	APB1
0x40001000～0x400013FF	TIM6 定时器	APB1
0x40001400～0x40001BFF	TIM7 定时器	APB1
0x40002400～0x400027FF	LCD	APB1
0x40002B00～0x40002BFF	RTC	APB1
0x40002C00～0x40002FFF	WWDG 窗口看门狗	APB1
0x40003000～0x400033FF	IWDG 独立看门狗	APB1
0x40003800～0x40003BFF	SPI2	APB1
0x40004400～0x400047FF	USART2	APB1
0x40004800～0x40004BFF	USART3	APB1
0x40005400～0x400057FF	I2C1	APB1
0x40005800～0x40005BFF	I2C2	APB1
0x40005C00～0x40005FFF	USB 寄存器	APB1
0x40006000～0x400061FF	USB 512B	APB1
0x40007000～0x400073FF	PWR 电源控制	APB1
0x40007400～0x400077FF	DAC	APB1
Ox40007C00～0x40007EFF	COMP＋RI	AHB
0x40010400～0x400107FF	EXTI	APB2
0x40010800～0x40010BFF	TIM9	APB2
0x40010C00～0x40010FFF	TIM10	APB2
0x40011000～0x400113FF	TIM11	APB2
0x40012400～0x400127FF	ADC	APB2
0x40013000～0x400133FF	SPI1	APB2
0x40013800～0x40013BFF	USART1	APB2
0x40020000～0x400203FF	GPIO 端口 A	AHB
0x40020400～0x400207FF	GPIO 端口 B	AHB

续表

地　　址	外　　设	总线
0x40020800～0x40020BFF	GPIO 端口 C	AHB
0x40020C00～0x40020FFF	GPIO 端口 D	AHB
0x40021400～0x400217FF	GPIO 端口 H	AHB
0x40023000～0x400233FF	CRC	AHB
0x40023800～0x40023BFF	RCC 复位和始终控制	AHB
0x40023C00～0x40023FFF	闪存存储器接口	AHB
0x40026000～0x400263FF	DMA	AHB

Chapter 3

第3章　Cortex-M3处理器的指令系统

【导读】 Cortex-M3处理器支持的指令集包括ARMv6的大部分16位Thumb指令和ARMv7的Thumb-2指令集。本章给出了指令的一般格式，并详细说明了存储器访问指令、数据处理指令、乘法除法指令、位操作指令和分支控制指令等使用，最后在说明符号定义伪指令、数据定义伪指令、汇编控制伪指令和宏指令的使用方法后，举例说明了汇编程序的编写方法。目前嵌入式开发主要以高级语言为主，汇编语言作为性能调优和底层初始化使用，本章介绍的指令集旨在让读者了解指令的种类、功能和基本使用方法，读懂汇编程序代码。

3.1　Cortex-M3处理器的指令系统

3.1.1　指令系统基本概念

1. 指令和指令集

冯·诺依曼体系结构采用存储程序的原则，即事先将程序存储到计算机中，程序是由计算机可执行的指令组成的，计算机的控制器根据程序指令控制计算机自动运行，即计算机执行程序的过程就是执行一条条指令的过程。

指令是指CPU能直接识别并执行的控制命令，它的表现形式是二进制编码。指令通常由操作码和操作数两部分组成，操作码指出该指令所要完成的操作，即指令的功能，操作数指出参与运算的数据或其存储地址等。

【思考题：为何指令中要使用数据存储地址而非直接使用数据?】

由于指令与CPU紧密相关，不同类型的CPU所对应的指令也有所不同，一台计算机所能执行的各种不同指令的全体，称为计算机的指令系统。一般同一系列的CPU指令集具有兼容性，即新的指令系统必须包括先前同系列CPU的指令系统，这样先前开发的各类程序在新CPU上才能正常运行。

机器语言是用来直接描述指令的最佳语言，是CPU能直接识别的唯一一种计算机语言，但机器语言书写大规模程序不容易维护。汇编语言使用助记符来代替和表示特定机器语言操作，相对机器语言更容易理解和维护，且汇编语言中可使用标签和符号等代表操作数或功能模块，使得程序更加灵活。一般汇编语言和其特定的机器语言描述的指令集是一一

对应的,但 CPU 无法识别汇编语言,因此需要汇编器进行转换。汇编语言目前已不像十几年前被广泛用于程序设计,只在操作系统、驱动程序等底层硬件操作和极高要求的程序优化场合下使用。

指令按照不同的分类可划分不同的种类。按照指令实现的功能,指令可分为:数据传送指令、算术运算指令、逻辑运算指令、程序控制指令、系统控制指令等。数据传送指令用来实现主存与 CPU 寄存器以及寄存器与寄存器之间的数据传输,例如 Thumb-2 的取一个字到寄存器指令 LDR、寄存器数据存储到主存指令 STR,寄存器间数据传送指令 MOV 等。算术运算是计算机能够执行的基本数值计算,算术运算指令包括加法 ADD、减法 SUB、乘法 MUL、除法 DIV 等指令。逻辑运算是对数据进行逻辑操作,包括逻辑与 AND、逻辑或 ORR、逻辑非 MVN 等三种基本操作以及同或、异或等组合逻辑操作。程序控制指令主要包括:转移指令 BL、断点指令 BKRT、分支指令 IT 等。系统控制指令包括休眠指令 WFI、WFE、空操作指令 NOP、开关中断指令 CPSIE、CPSID 等。

按照数据存取方式,指令可分为寄存器-寄存器型、寄存器-存储器型和存储器-存储器型。寄存器-寄存器型将数据存放在寄存器中进行操作,例如 Cortex-M3 的大多数数据传输和算术逻辑运算指令,寄存器-寄存器型指令编码简单、执行速度快,指令周期相近。寄存器-存储器型指令可直接对存储器操作数进行访问,如访存操作 LDR、STR 等,指令周期相差较大,执行速度较慢。存储器-存储器型无需寄存器保存数据,其执行的访存较多,执行速度慢,Cortex-M3 绝大多数指令不采用存储器-存储器结构。

2. Cortex-M3 指令的编码方式

Cortex-M3 支持 16 位和 32 位的 Thumb-2 指令集,一个典型的 16 位指令的编码如图 3-1 所示。

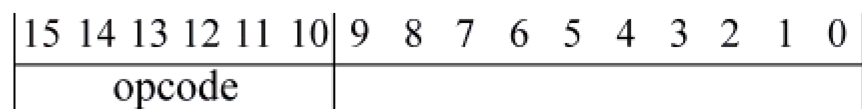


图 3-1 Thumb-2 16 位指令编码格式

16 位指令的操作码部分通过 6 个 bit 进行分类,如表 3-1 所示,每一类指令根据具体情况进行二次编码,因此,Cortex-M3 的指令操作码部分是不等长的,但可以通过多级译码实现,每一级译码的操作码是等长的,由此实现了指令的灵活性和复杂性的均衡。

表 3-1 opcode 分类定义

操作码 opcode	指令或指令类
00xxxx	立即数寻址移位、加法、减法、送数和比较指令
010000	数据处理指令
010001	特殊的数据、分支和交换指令
01001x	寄存器偏移寻址加载指令
0101xx 011xxx 100xxx	单数据加载/存储指令

续表

操作码 opcode	指令或指令类
10100x	PC 相关寻址类指令
10101x	SP 相关寻址类指令
1011xx	16 位的其他指令
11000x	多寄存器存储治理，如 STM，STMIA，STMEA
11001x	多寄存器加载指令，如 LDM，LDMIA，LDMFD
1101xx	有条件转移、SVC 指令
11100x	无条件转移指令

对于 32 位指令，高 16 位的操作码 opcode 的取值为 11101、11110 和 11111，此时处理器会将下一个 16 位和当前 16 位组合成一个 32 位指令，如图 3-2 所示（即 op1 的取值为 01，10 和 11，当 op1 为 00 时，表明是一条 16 位指令）。

15 14 13	12 11	10 9 8 7 6 5 4	3 2 1 0	15	14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1	op1	op2		op	

图 3-2　Thumb-2 32 位指令的编码格式

两个寄存器数据的逻辑与 ADD 和逻辑或 ORR 指令编码如图 3-3 所示，从表 3-1 中可以看出逻辑与和逻辑或属于数据处理指令类，数据处理指令又采用了 4 个比特进行了二次编码。

15 14 13 12 11 10	9 8 7 6	5 4 3	2 1 0
0 1 0 0 0 0	0 0 0 0	Rm	Rdn

15 14 13 12 11 10	9 8 7 6	5 4 3	2 1 0
0 1 0 0 0 0	1 1 0 0	Rm	Rdn

图 3-3　逻辑与和逻辑或的 16 位指令编码

【**思考题**：Thumb2 指令编码是否违反了 RISC 设计思想?】

3.1.2　指令格式

Cortex-M3 不支持 ARM 指令集，支持的指令集包括 ARMv6 的大部分 16 位 Thumb 指令和 ARMv7 的 Thumb-2 指令集。Thumb-2 指令集是一个 16/32 位混合的指令系统。

指令的一般格式如下：

```
<opcode>{<cond>}{S}{.N|.W}<Rd>,<Rn>{,<operand2>}
```

- opcode 是操作码，如 ADD、LDR 和 STR 等，规定所执行的具体操作；

- cond 是可选的条件码，如 EQ、NE 和 CS 等，规定指令执行所满足的条件，条件码的说明见表 3-2；
- S 是可选后缀，若指定 S，则需要根据指令执行结果去更新程序状态寄存器 xPSR 相应的标志位；
- .N 表示 16 位指令，.W 表示 32 位指令，默认为 16 位指令；
- Rd 是目标寄存器；
- Rn 是存放第 1 个操作数的寄存器；
- operand2 是第 2 个操作数。

这里，符号{ }和＜ ＞的含义如下：

- { }表示可选的，例如{＜cond＞}表示条件码是可选的，可以有条件码也可无条件码；
- ＜ ＞表示必需的，例如＜Rd＞表示必须有目标寄存器。

表 3-2　条件码定义

后　缀	标　志	含　义
EQ	Z=1	相等
NE	Z=0	不相等
CS or HS	C=1	无符号数大于或等于
CC or LO	C=0	无符号数小于
MI	N=1	负的
PL	N=0	正的或为 0
VS	V=1	溢出
VC	V=0	无溢出
HI	C=1 and Z=0	无符号数大于
LS	C=0 or Z =1	无符号数小于或等于
GE	N=V	有符号数大于或等于
LT	N!=V	有符号数小于
GT	Z =0 and N=V	有符号数大于
LE	Z =1 or N!=V	有符号数小于或等于
AL		无条件执行

条件执行是 ARM 处理器的一个优化程序速度的典型方式，可以减少不必要的跳转。如图 3-4 所示的 C 语言代码的 ARM 汇编结果：

```
if(a>b)
    a++
else
    b++
```

⟹

```
CMP R0, R1
ADDHI R0, R0, #1
ADDLS R1, R1, #1
```

图 3-4　条件执行汇编指令

3.1.3　寻址方式

寻址方式是根据指令中给出的操作数字段来实现寻找真实操作数的方式，Cortex-M3支持 8 种寻址方式。

1）寄存器寻址

操作数的值在寄存器中，指令中的地址码字段给出的是寄存器编号，寄存器的内容是操作数，指令执行时直接取出寄存器值操作，例如：

```
MOV   R1,R2          ;R1←R2
SUB   R0,R1,R2       ;R0←R1-R2
```

2）立即数寻址

数据就包含在指令当中，立即寻址指令的操作码字段后面的地址码部分就是操作数本身，取出指令也就取出了可以立即使用的操作数（也称为立即数）。立即数要以“#”为前缀，表示十六进制数值时以“0x”表示，例如：

```
ADD   R0,R0,#1       ;R0←R0+1
MOV   R0,#0xff00     ;R0←0xff00
```

3）寄存器移位寻址

寄存器移位寻址是把第 2 个寄存器操作数移位之后送给第 1 个操作数，例如：

```
MOV   R0,R2,LSL #3        ;R2的值左移 3位,结果放入 R0,即 R0=R2X8。
AND   R1,R1,R2,LSL R3     ;R2的值左移 R3位,然后和 R1相与操作,结果放入 R1。
```

可采用的移位操作如下：

- LSL：逻辑左移（Logical Shift Left），寄存器中字的低端空出的位补 0。
- LSR：逻辑右移（Logical Shift Right），寄存器中字的高端空出的位补 0。
- ASR：算术右移（Arithmetic Shift Right），移位过程中保持符号位不变，即如果源操作数为正数，则字的高端空出的位补 0，否则补 1。
- ROR：循环右移（Rotate Right），由字的低端移出的位填入字的高端空出的位。
- RRX：带扩展的循环右移（Rotate Right extended by 1 place），操作数右移一位，高端空出的位用进位标志 C 的值填充。

4）寄存器间接寻址

指令中的地址码给出的是一个通用寄存器编号，所需要的操作数保存在该寄存器的值作为地址的存储单元中，即寄存器为操作数的地址指针，操作数存放在存储器中，例如：

```
LDR   R0,[R1]   ;R0←[R1](将 R1的值作为地址,取出此地址中的数据保存在 R0中)
STR   R0,[R1]   ;[R1]←R0(将 R0的值写入到以 R1的值为地址的存储器空间中)
```

5）变址寻址

变址寻址是将基址寄存器的内容与指令中给出的偏移量相加，形成操作数的有效地址，

变址寻址用于访问基址附近的存储单元,常用于查表,数组操作,外设控制器的内部寄存器访问等,例如:

```
LDR   R2,[R3,#4]        ;R2←[R3+4](将 R3 的数值加 4 作为地址,取出此地址的数值保存在 R2 中)
STR   R1,[R0,#-2]       ;[R0-2]←R1(将 R0 的数值减 2 作为地址,把 R1 中的内容保存到此地址的
                        ;存储单元中)
```

6) 多寄存器寻址

采用多寄存器寻址方式,一条指令可以完成多个寄存器值的传送,这种寻址方式用一条指令最多可以完成 16 个寄存器值的传送,例如:

```
LDMIA   R0,{R1,R2,R3,R5}    ;将 R0,R0+4,R0+8,R0+12 地址处的数据分别送到寄存器 R1,R2,
                            ;R3 和 R5 中,R0 的值保持不变
STMIA   R0!,{R1-R7}         ;将 R1~R7 的数据保存到存储器中,存储器指针 R0 在保存第一个值
;之后增加,增长方向为向上增长,即 R1~R7 的值存储在 R0,R0+4,R0+8,R0+12,R0+16,R0+20,R0
;+24 的地址中,指令执行完后,R0 的值变成 R0+24
STMDA   R0!,{R1-R7}         ;将 R1~R7 的数据保存到存储器中,存储器指针 R0 在保存第一个值
;之后减少,增长方向为向下增长,即 R1~R7 的值存储在 R0-24,R0-20,R0-16,R0-12,R0-8,R0-4,
;R0 的地址中,指令执行完后,R0 的值变成 R0-24
```

7) 栈寻址

栈寻址是通过栈指针 R13 进行数据读写的方式,如 POP 和 PUSH 操作等,其数据存储和加载的地址由 SP 寄存器隐含给出,例如:

```
PUSH   {R0-R3}              ;将 R0,R1,R2,R3 四个寄存器的值压入栈中
POP     {R0-R2}             ;将栈顶的数据依次读取到 R0,R1,R2 中
LDMIA  SP!,{R1,R2,R3,R5}    ;将栈顶的数据依次读入到 R1,R2,R3,R5 中
```

8) 相对寻址

相对寻址是变址寻址的一种变通,由程序计数器 PC 提供基准地址,指令中的地址码字段作为偏移量,两者相加后得到的地址即为操作数的有效地址,例如:

```
    LDR   R2,[PC,#4]        ;R2←[PC+4](将 PC 加 4 作为地址,取出此地址的数保存在 R2 中)
    BL    ROUTE1            ;调用到 ROUTE1 子程序,等价于 LDR   PC,[PC,#6]
    BEQ   LOOP              ;条件跳转到 LOOP 标号处,等价于 LDR   PC,[PC,#2]
LOOP
    MOV R2,#2
ROUTE1
    MOV R1,#3
```

3.1.4 数据传送指令

最基本的数据传送指令是寄存器间的数据传送,此外,还包括立即数加载到寄存器,特殊寄存器的读写指令等。数据传送指令包括 MOV、MVN、MRS 和 MSR 等,具体指令格式

及其功能如表 3-3 所示。

表 3-3 数据传送指令

示 例	功 能 描 述
MOV <Rd>, #<immed_8>	将 8 位立即数传到目标寄存器
MOV <Rd>, <Rn>	将寄存器值传给低目标寄存器
MVN <Rd>, <Rm>	寄存器值取反后传给目标寄存器
MOV{S}. W <Rd>, #<immed_12>	将 12 位立即数传送到寄存器中
MOV{S}. W <Rd>, <Rm>{,<shift>}	将移位后的寄存器值传到寄存器
MOVT. W <Rd>, #<immed_16>	将 16 位立即数传送到寄存器的高半字[31:16]
MOVW. W <Rd>, #<immed_16>	16 位立即数传到寄存器的低半字[15:0],将高半字[31:16]清零
MRS <Rd>, <SReg>	读特殊功能寄存器 SReg 到寄存器 Rd
MSR <SReg>, <Rn>	写 Rn 到特殊功能寄存器 SReg

例如:

```
MOV   R0, #8               ;R0=8
MOV   R1,  R0              ;R1=R0=8
MVN   R2,  R1              ;R2=0xFFFFFFF7
MOV.W   R4,  R0 LSL, #2    ;R4=32
MOVW.W   R1,#0x1234        ;R1 =0x1234
MOVT.W   R1,#0x5678        ;R1 =0x56781234
MRS   R1  APSR             ;将 R1 的值写入到状态寄存器 APSR
```

3.1.5 存储器访问指令

存储器访问指令包括存储器寄存器传输指令 LDR、STR;多寄存器加载指令 LDM、多寄存器存储指令 STM 以及压栈指令 PUSH 和出栈指令 POP 等。

1) 加载指令 LDR 和存储指令 STR

LDR 实现从存储器中加载操作数到寄存器,STR 实现从寄存器存储数据到存储器。LDR 和 STR 的语法格式为:

语法格式:

```
op {type}{cond} Rt, [Rn{, #offset}]     ;立即偏移
op {type}{cond} Rt, [Rn, #offset]!      ;前变址
op {type}{cond} Rt, [Rn], #offset       ;后变址
```

这里,op 是 LDR 或 STR。type 指定传送的是字节还是半字,缺省为传送一个字。cond 是可选条件码,Rt 是指定的加载或存储的寄存器,Rn 是存储器地址存放的寄存器,

offset 是偏移量。

type 可以是：

- B：传送无符号的字节；
- SB：传送有符号的字节；
- H：传送无符号的半字；
- SH：传送有符号的半字。

LDR 和 STR 指令进行数据加载和存储时，涉及三种寻址方式，分别为立即偏移的基址变址寻址方式、前变址的基址变址寻址方式和后变址的基址变址寻址方式。这里，基址寄存器为 Rn。

- 立即偏移的基址变址寻址方式，例如 LDR R0，[R1，#4]，这种寻址方式是将基址寄存器 R1 的内容和偏移量 4 相加形成操作数的有效地址；操作完成后，基址寄存器 R1 的内容不变。
- 前变址的基址变址寻址方式，例如 LDR R0，[R1，#4]!，这种寻址方式是将基址寄存器 R1 的内容和偏移量 4 相加形成操作数的有效地址；操作完成后，基址寄存器 R1 的内容更新为新的地址，即 R1＝R1＋4。
- 后变址的基址变址寻址方式，例如 LDR R0，[R1]，#4，这种寻址方式直接将基址寄存器 R1 的内容作为操作数的有效地址；操作完成后，基址寄存器 R1 的内容更新为原有内容和偏移量之和，即 R1＝R1＋4。

使用举例：

```
① LDR R6, [R10]
```

存储器操作数是寄存器间接寻址方式，直接将 R10 寄存器的内容作为操作数的有效地址，该指令实现将有效地址的存储器操作数加载到 R6 寄存器；

```
② LDRNE R6, [R5, #960]!
```

这是带条件码的指令，当程序状态寄存器的 Z 标志位为 0 时才执行该指令。存储器操作数是前变址的基址变址寻址方式，将 R5 寄存器的内容和偏移量 #960 相加作为操作数的有效地址，该指令实现将有效地址的存储器操作数加载到 R6 寄存器；操作完成后，R5 寄存器的内容增加 960。

```
③ STRH R6, [R4], #4
```

这是一个无符号半字的存储指令。存储器操作数是后变址的基址变址寻址方式，直接将 R4 寄存器的内容作为操作数的有效地址，该指令实现将有效地址的存储器操作数（无符号半字）加载到 R6 寄存器；操作完成后，R4 寄存器的内容增加 4。

如果需要对双字进行存取，指令的格式为：

```
opD {cond} Rt, Rt2, [Rn{, #offset}]      ;立即偏移，双字指令
opD {cond} Rt, Rt2, [Rn, #offset]!       ;前变址，双字指令
opD {cond} Rt, Rt2, [Rn], #offset        ;后变址，双字指令
```

其中，op 为 STM 或 LDR，Rt 和 Rt2 为双字存取的寄存器，Rn 为地址寄存器，例如：STRD R1，R2，[R8]，#-16，这是一个双字存储指令。存储器操作数是后变址的基址变址寻址方式，将 R1 的内容存储到 R8 寄存器所指定的有效地址中，将 R2 的内容存储到 R8 寄存器的内容+4 所指定的有效地址中；操作完成后，R8 寄存器的内容减少 16。存储器访问指令的指令格式和功能详见表 3-4。

表 3-4　存储器数据加载和存储指令

示　例	功 能 描 述
LDRB Rd，[Rn，#offset]	从地址 Rn+offset 处读取一个字节送到 Rd
LDRH Rd，[Rn，#offset]	从地址 Rn+offset 处读取一个半字送到 Rd
LDR Rd，[Rn，#offset]	从地址 Rn+offset 处读取一个字送到 Rd
LDRD Rd1，Rd2，[Rn，#offset]	从地址 Rn+offset 处读取一个双字(64 位整数)送到 Rd1(低 32 位)和 Rd2(高 32 位)中
STRB Rd，[Rn，#offset]	把 Rd 中的低字节存储到地址 Rn+offset 处
STRH Rd，[Rn，#offset]	把 Rd 中的低半字存储到地址 Rn+offset 处
STR Rd，[Rn，#offset]	把 Rd 中的低字存储到地址 Rn+offset 处
STRD Rd1，Rd2，[Rn，#offset]	把 Rd1(低 32 位)和 Rd2(高 32 位)表达的双字存储到 Rn+offset 处
LDR.W Rd，[Rn，#offset]!	字的带预索引加载
LDRB.W Rd，[Rn，#offset]!	字节的带预索引加载(不扩展符号位)
LDRH.W Rd，[Rn，#offset]!	半字的带预索引加载(不扩展符号位)
LDRD.W Rd1，Rd2，[Rn，#offset]!	双字的带预索引加载(不扩展符号位)
LDRSB.W Rd，[Rn，#offset]! LDRSH.W Rd，[Rn，#offset]!	字节/半字的带预索引加载，并且在加载后执行带符号扩展成 32 位整数
STR.W Rd，[Rn，#offset]!	字/字节/半字/双字的带预索引存储
STRB.W Rd，[Rn，#offset]!	字节的带预索引存储
STRH.W Rd，[Rn，#offset]!	半字的带预索引存储
STRD.W Rd1，Rd2，[Rn，#offset]!	双字的带预索引存储

2）批量加载指令 LDM 和批量存储指令 STM

LDM 实现由基址寄存器指示的一片连续存储器中的数据批量加载到多个寄存器，STM 实现将多个寄存器中的数据批量存储到由基址寄存器指示的一片连续存储器。

语法格式：

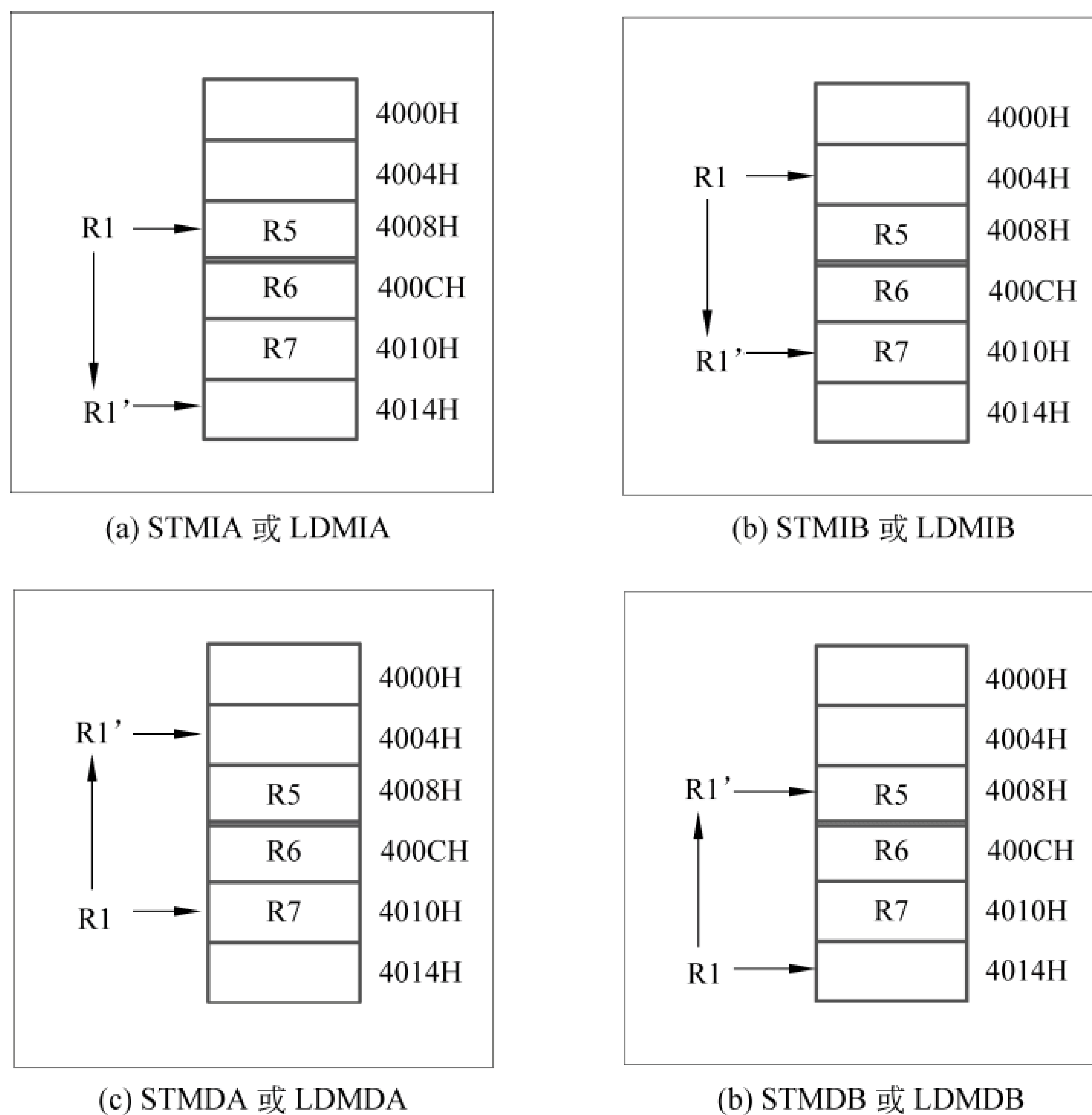
```
op {addr_mode} {cond} Rn{!}, reglist
```

这里，op 是 LDM 或 STM，addr_mode 是地址变化模式，cond 是可选条件码，Rn 是基址寄存器，! 为可选的回写后缀，若选用该后缀，则数据传送完毕后，将最后的地址写入基址寄

存器，reglist 是多个数据加载或存储的寄存器列表。

addr_mode 可以取下列值：

- IA(Increment After)意味着在每一次访问之后地址递增，如图 3-5(a)所示。
- IB(Increment Before)意味着每一次访问之前地址递增，如图 3-5(b)所示。
- DA(Decrement After)意味着每一次访问之后地址递减，如图 3-5(c)所示。
- DB(Decrement Before)意味着在每一次访问之前地址递减，如图 3-5(d)所示。



(a) STMIA 或 LDMIA　　(b) STMIB 或 LDMIB

(c) STMDA 或 LDMDA　　(b) STMDB 或 LDMDB

图 3-5　批量加载和批量存储指令中的地址模式

指令举例如下：

```
① LDM R8, {R0,R2,R9}
```

该指令实现将 R8 指示的存储器的连续内容加载到三个寄存器 R0、R2 和 R9。R8 的值保持不变。

```
② STMDB R1!,{R3-R6,R11,R12}
```

该指令实现将 R3-R6、R11 和 R12 寄存器中的内容存储到 R1 指示的存储器中，并将最后一个存储单元的地址回写到 R1 寄存器。

在多寄存器存储指令中，基址寄存器不能是 PC，寄存器列表中不能含有 SP；若是 STM 指令，寄存器列表一定不能含有 PC；若是 LDM 指令，若寄存器列表中含有 LR 则一定不能

含有 PC;如果增加回写后缀,则寄存器列表中一定不能含有基址寄存器。多寄存器存储指令的指令格式和功能详见表 3-5。

表 3-5　多寄存器访存指令

示　　例	功 能 描 述
LDMIA Rd!,{寄存器列表}	16 位指令,从 Rd 处读取多个字,并依次送到寄存器列表中的寄存器。每读一个字后 Rd 自增一次
STMIA Rd!,{寄存器列表}	16 位指令,存储寄存器列表中各寄存器的值依次存储到 Rd 给出的地址。每存一个字后 Rd 自增一次
LDMIA. W Rd!,{寄存器列表}	32 位指令,从 Rd 处读取多个字,并依次送到寄存器列表中的寄存器。每读一个字后 Rd 自增一次
LDMDB. W Rd!,{寄存器列表}	32 位指令,从 Rd 处读取多个字,并依次送到寄存器列表中的寄存器。每读一个字前 Rd 自减一次
STMIA. W Rd!,{寄存器列表}	32 位指令,依次存储寄存器列表中各寄存器的值到 Rd 给出的地址。每存一个字后 Rd 自增一次
STMDB. W Rd!,{寄存器列表}	32 位指令,存储多个字到 Rd 处。每存一个字前 Rd 自减一次

3) 压栈指令 PUSH 和出栈指令 POP

当栈指针指向最后压入栈的数据时,称为满栈(Full Stack);当栈指针指向下一个将要放入数据的空位置时,称为空栈(Empty Stack)。压栈时,栈由低地址向高地址生长时,称为递增栈(Ascending Stack),栈由高地址向低地址生长时,称为递减栈(Descending Stack)。由此可以组合四种栈模型:递增满栈、递增空栈、递减满栈、递减空栈。Cortex-M3 使用的是向下生长的满栈模型。栈指针 SP 指向最后一个被压入栈的 32 位数值。在下一次压栈时,SP 先自减 4,再存入新的数值,如图 3-6 所示。

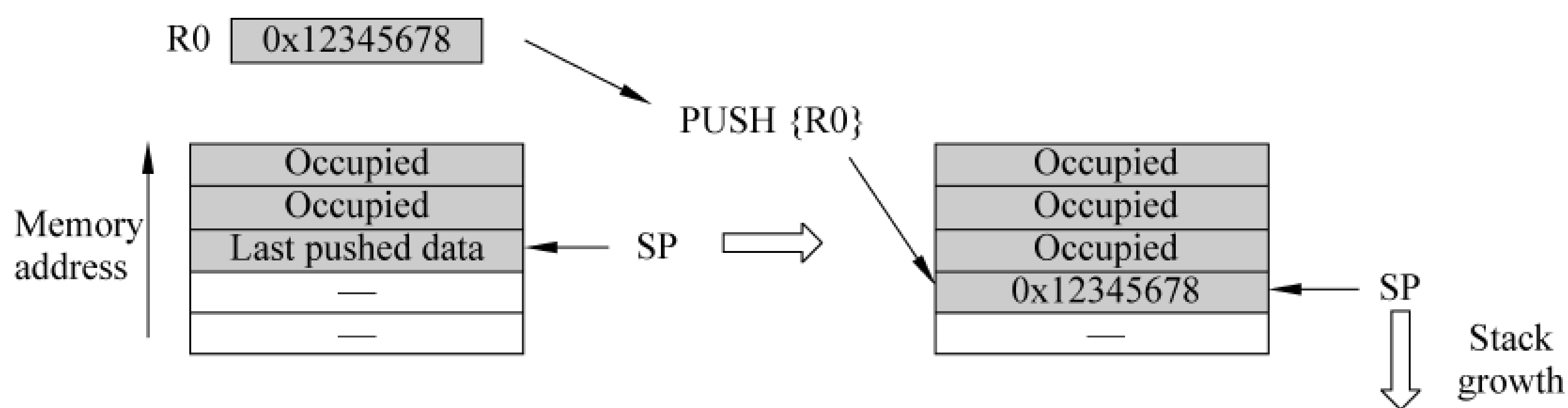


图 3-6　压栈操作过程

出栈操作刚好相反:先从 SP 指针处读出上一次被压入的值,再把 SP 指针自增 4,如图 3-7 所示。

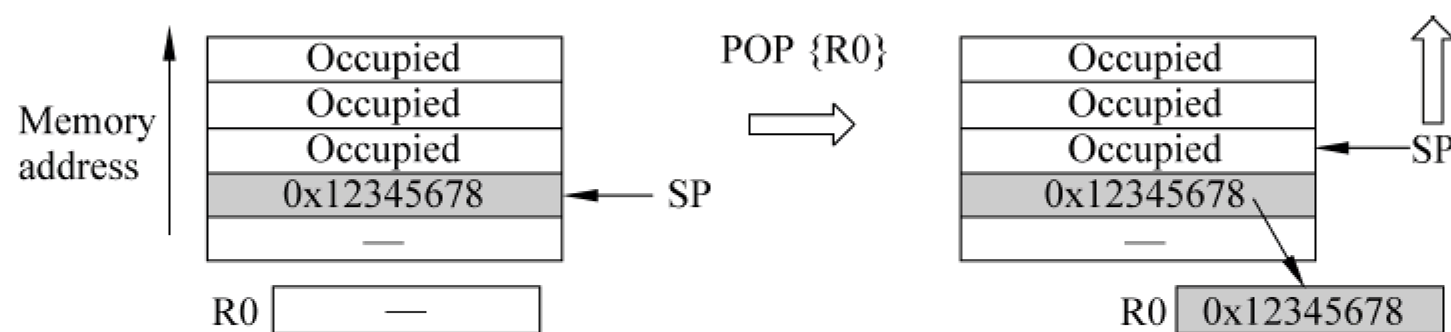


图 3-7　出栈操作过程

PUSH 实现以递减满栈方式的压栈操作；POP 实现以递减满栈方式的出栈操作。因此，PUSH 相当于指令 STMDB 的功能，POP 相当于指令 LDMIA 的功能。

语法格式：

```
PUSH {cond} reglist
POP  {cond} reglist
```

这里，cond 是可选条件码；reglist 是压栈或出栈需要的寄存器列表。

寄存器列表中一定不能含有 SP；对于 PUSH 指令，寄存器列表一定不能含有 PC；对于 POP 指令，如果寄存器列表中含有 LR，则一定不能含有 PC。

使用举例：

```
① PUSH {R0, R4-R7}
```

该指令实现将寄存器列表 R0、R4、R5、R6 和 R7 中数据压入栈。

```
② POP {R0,R6,PC}
```

该指令实现将寄存器列表 R0、R6 和 PC 中的数据弹出栈。

3.1.6 算术运算指令

算术运算指令主要包括加减乘除操作，其指令格式和功能如表 3-6 所示。

表 3-6 算术运算指令

示例		功能描述
ADD Rd, Rn, Rm	; Rd = Rn+Rm	常规加法
ADD Rd, Rm	; Rd += Rm	
ADD Rd, #imm	; Rd += imm	im8(16 位指令)或 im12(32 位指令)
ADC Rd, Rn, Rm	; Rd = Rn+Rm+C	带进位的加法， Im8 或 im12
ADC Rd, Rm	; Rd += Rm+C	
ADC Rd, #imm	; Rd += imm+C	
SUB Rd, Rn	; Rd -= Rn	常规减法
SUB Rd, Rn, #imm3	; Rd = Rn−imm3	
SUB Rd, #imm8	; Rd -= imm8	
SUB Rd, Rn, Rm	; Rd = Rn−Rm	
SBC Rd, Rm	; Rd -= Rm+C	带借位的减法
SBC. W Rd, Rn, #imm12	; Rd = Rn−imm12−C	
SBC. W Rd, Rn, Rm	; Rd = Rn−Rm−C	

续表

示　　例		功 能 描 述
RSB.W Rd，Rn，#imm12	；Rd ＝ imm12－Rn	反向减法
RSB.W Rd，Rn，Rm	；Rd ＝ Rm－Rn	
MUL Rd，Rm	；Rd *＝ Rm	常规乘法
MUL.W Rd，Rn，Rm	；Rd ＝ Rn * Rm	
MLA Rd，Rm，Rn，Ra	；Rd ＝ Ra＋Rm * Rn	乘加
MLS Rd，Rm，Rn，Ra	；Rd ＝ Ra－Rm * Rn	乘减
UDIV Rd，Rn，Rm	；Rd ＝ Rn/Rm	无符号除法，硬件支持的除法，余数被丢弃
SDIV Rd，Rn，Rm	；Rd ＝ Rn/Rm	带符号除法，硬件支持的除法，余数被丢弃
SMULL RL，RH，Rm，Rn	；[RH：RL]＝ Rm * Rn	带符号的 64 位乘法
SMLAL RL，RH，Rm，Rn	；[RH：RL]＋＝ Rm * Rn	
UMULL RL，RH，Rm，Rn	；[RH：RL]＝ Rm * Rn	无符号的 64 位乘法
UMLAL RL，RH，Rm，Rn	；[RH：RL]＋＝ Rm * Rn	

1）ADD、ADC、SUB、SBC 和 RSB

- ADD 实现第 2 个操作数 Operand2 或立即数 imm 与第 1 个操作数 Rn 相加，并将结果送至 Rd 目标寄存器。
- ADC 实现第 2 个操作数 Operand2 与第 1 个操作数 Rn 以及进位标志相加，并将结果送至 Rd 目标寄存器。
- SUB 实现第 1 个操作数 Rn 与第 2 个操作数 Operand2 或立即数 imm 相减，并将结果送至 Rd 目标寄存器。
- SBC 实现第 1 个操作数 Rn 与第 2 个操作数 Operand2 相减，再减去 C 条件标志位的反码，并将结果送至 Rd 目标寄存器。
- RSB 实现第 2 个操作数 Operand2 与第 1 个操作数 Rn 相减，并将结果送至 Rd 目标寄存器。

使用举例：

① ADD R2, R1, R3

该指令实现 R1 寄存器和 R3 寄存器的内容相加，并将结果送至 R2 寄存器，即 R2＝R1＋R3。

② SUBS R7, R5, #256

该指令实现 R5 寄存器的内容与立即数#256 相减，并将结果送至 R7 寄存器，即 R7＝R5-256。同时，根据运算结果影响标志位。

③ RSB R8, R8, #240

该指令实现立即数 240 与 R8 寄存器的内容相减，并将结果送至 R8 寄存器，即 R8=240-R8。

④ ADDHI R10, R0, R3

当 C=1 且 Z=0 成立时，执行该指令。该指令实现将 R0 寄存器与 R3 寄存器的内容以及进位标志相加，并将结果送至 R10 寄存器，即 R10=R0+R3+C。

2）MUL、MLA 和 MLS

- MUL 指令实现将 Rn 寄存器和 Rm 寄存器的内容相乘，并将结果存入 Rd 寄存器。
- MLA 指令实现将 Rn 寄存器和 Rm 寄存器的内容相乘，并将乘法结果和 Ra 寄存器的内容相加，再将最终结果存入 Rd 寄存器。
- MLS 指令实现将 Rn 寄存器和 Rm 寄存器的内容相乘，并将 Ra 寄存器中的内容与乘法结果相减，再将最终结果存入 Rd 寄存器。

使用举例：

① MUL R7, R2, R5

该指令实现将 R2 寄存器的内容和 R5 寄存器的内容相乘，然后将乘法结果存入 R10 寄存器，即 R10=R2 * R5。

② MLA R9, R3, R4, R6

该指令实现将 R3 寄存器的内容和 R4 寄存器的内容相乘，然后将乘法结果再与 R6 寄存器内容相加，最后将计算结果存入 R9 寄存器，即 R9=R3 * R4+R6。

③ MLS R10, R5, R6, R7

该指令实现将 R5 寄存器的内容和 R6 寄存器的内容相乘，然后将 R7 寄存器的内容与乘法结果相减，最后将计算结果存入 R10 寄存器，即 R10=R7-R5 * R6。

3）SDIV 和 UDIV

- SDIV 实现有符号整数相除，将 Rn 寄存器的内容与 Rm 寄存器的内容相除，计算结果存入 Rd 寄存器。
- UDIV 实现无符号整数相除，将 Rn 寄存器的内容与 Rm 寄存器的内容相除，计算结果存入 Rd 寄存器。

除法指令同样不能使用 SP 或 PC，对条件标志位没有影响。

使用举例：

① SDIV R0, R1, R2

该指令实现 R1 寄存器内容与 R2 寄存器内容的有符号数相除，计算结果存入 R0 寄存器，即 R0=R1/R2。

② UDIV R7, R7, R3

该指令实现 R1 寄存器内容与 R2 寄存器内容的有无号数相除,计算结果存入 R7 寄存器,即 R7=R7/R3。

3.1.7 逻辑运算指令

逻辑运算指令包括与、或、非基本操作及其扩展,其指令格式及功能描述见表 3-7 所示。

表 3-7 逻辑运算指令

示 例	功 能 描 述
AND Rd, Rn ; Rd &= Rn AND.W Rd, Rn, #imm12 ; Rd = Rn & imm12 AND.W Rd, Rm, Rn ; Rd = Rm & Rn	按位与
ORR Rd, Rn ; Rd |= Rn ORR.W Rd, Rn, #imm12 ; Rd = Rn | imm12 ORR.W Rd, Rm, Rn ; Rd = Rm | Rn	按位或
BIC Rd, Rn ; Rd &= ~Rn BIC.W Rd, Rn, #imm12 ; Rd = Rn & ~imm12 BIC.W Rd,Rm, Rn ; Rd = Rm & ~Rn	位清零 Rn 与 Operand2 的反码按位逻辑与
ORN.W Rd, Rn, #imm12 ; Rd = Rn | ~imm12 ORN.W Rd, Rm, Rn ; Rd = Rm | ~Rn	按位或反码
EOR Rd, Rn ; Rd ^= Rn EOR.W Rd, Rn, #imm12 ; Rd = Rn ^ imm12 EOR.W Rd, Rm, Rn ; Rd = Rm ^ Rn	按位异或

- AND 实现第 1 个操作数 Rm 与第 2 操作数 Operand2 的逻辑与操作。
- ORR 实现第 1 个操作数 Rm 与第 2 操作数 Operand2 的逻辑或操作。
- EOR 实现第 1 个操作数 Rm 与第 2 操作数 Operand2 的逻辑异或操作。
- BIC 实现第 1 个操作数 Rm 与第 2 操作数 Operand2 的反码的逻辑与操作,一般可用来清除第 1 个操作数 Rm 的相应位。
- ORN 实现第 1 个操作数 Rm 与第 2 操作数 Operand2 的反码的逻辑或操作。

使用举例:

```
① AND R1, R1, #0x00FF
```

该指令实现 R1 寄存器的内容与立即数#0x00FF 进行逻辑与操作,并将结果送至 R1 寄存器,即实现 R1 寄存器的高 16 位清零。

```
② BIC R2, R2, #0x000B
```

该指令实现 R2 寄存器的内容与立即数#0x000B 的反码进行逻辑与操作,并将结果送至 R2 寄存器,即实现 R2 寄存器的第 0、1 和 3 位的清零。

逻辑运算中逻辑非的操作由送数指令 MVN 实现。

3.1.8 移位和循环指令

移位运算包括逻辑移位、算术移位以及循环移位等特殊形式,具体指令格式和功能见表 3-8。

表 3-8 移位和循环指令

示　例		功能描述
LSL Rd, Rn, #imm5 LSL Rd, Rn LSL.W Rd, Rm, Rn	; Rd = Rn<<imm5 ; Rd <<= Rn ; Rd = Rm<<Rn	逻辑左移
LSR Rd, Rn, #imm5 LSR Rd, Rn LSR.W Rd, Rm, Rn	; Rd = Rn>>imm5 ; Rd >>= Rn ; Rd = Rm>>Rn	逻辑右移
ASR Rd, Rn, #imm5 ASR Rd, Rn ASR.W Rd, Rm, Rn	; Rd = Rn·>>imm5 ; Rd ·>> = Rn ; Rd = Rm·>> Rn	算术右移
ROR Rd, Rn ROR.W Rd, Rm, Rn	; Rd >> = Rn ; Rd = Rm >> Rn	循环右移
RRX.W Rd, Rn	; Rd=(Rn>>1)+(C<<31)	带进位的右移一位

- ASR 算术右移指令实现对 Rm 寄存器中的数进行右移 Rn 或 imm5 位操作,左端使用第 31 位值来补充,如图 3-8(a)所示。
- LSL 逻辑左移指令实现对 Rm 寄存器中的数进行左移 Rn 或 imm5 位操作,低位使用 0 填充,如图 3-8(b)所示。
- LSR 逻辑右移指令实现对 Rm 寄存器中的数进行右移 Rn 或 imm5 位操作,左端使用 0 填充,如图 3-8(c)所示。
- ROR 循环右移指令实现对 Rm 寄存器中的数进行右移 Rn 或 imm5 位操作,左端使用右端移出的位来进行填充,如图 3-8(d)所示。
- RRX 带进位的循环右移,左端使用进位标志 C 来进行填充,如图 3-8(e)所示。

使用举例:

① ASR R1, R2, #9

该指令实现将 R2 寄存中的内容算术右移 9 位(左端使用第 31 位值来填充),然后将结果填入 R1 寄存器。

② LSLS R3, R4, #3

该指令附加标志位 S,计算结果可能影响标志位。该指令实现将 R4 寄存器逻辑左移 3 位(低位使用 0 值来填充),然后将结果填入 R3 寄存器。

③ ROR R5, R6, #6

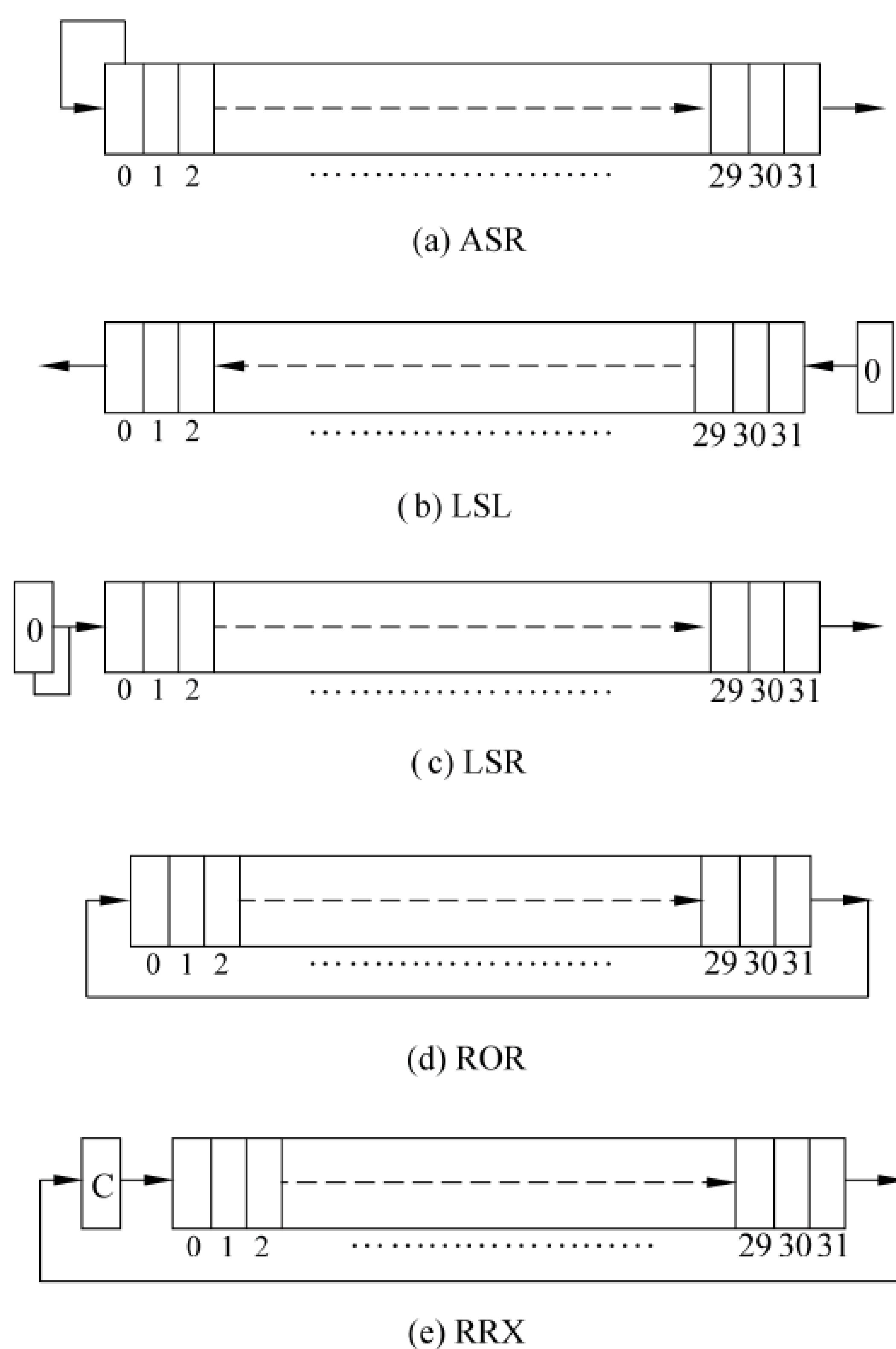


图 3-8　移位操作

该指令实现将 R6 寄存器循环右移 6 位(左端使用右端移出的位进行填充),然后将结果填入 R5 寄存器。

3.1.9　比较指令

CMP 和 CMN 是比较操作指令,其具体指令格式和功能见表 3-9。

表 3-9　比较指令

示　　例	功 能 描 述
CMN ＜Rn＞,＜Rm＞	将 Rm 取二进制补码后再与 Rn 比较
CMP ＜Rn＞,#＜immed_8＞	Rn 与 8 位立即数比较,并根据结果更新标志位的值
CMP ＜Rn＞,＜Rm＞	Rn 与 Rm 比较,并根据结果更新标志位的值
CMN. W ＜Rn＞,#＜ immed_12＞	Rn 与 12 位立即数取补后的值比较
CMN. W ＜Rn＞,＜Rm＞{,＜shift＞}	Rn 与移位后的 Rm 取补后的值比较
CMP. W ＜Rn＞,#＜ immed_12＞	Rn 与 12 位立即数比较
CMP. W ＜Rn＞,＜Rm＞{,＜shift＞}	Rn 与按需移位后的 Rm 比较,Rm 的值不变

CMP 指令实现将 Rn 寄存器的内容减去第 2 个操作数 Operand2，计算结果影响标志位。该指令类似于 SUBS 指令，所不同的是 CMP 指令丢弃减法计算结果，而 SUBS 指令需要保存减法计算结果。

CMN 指令实现将第 2 个操作数加到 Rn 寄存器中，计算结果影响标志位。该指令类似于 ADDS 指令，所不同的是 CMN 指令丢弃加法计算结果，而 ADDS 指令需要保存加法计算结果。

比较指令根据计算结果影响标志位 N、Z、C 和 V。

使用举例：

① CMP R1, #6400

该指令将 R1 寄存器的内容减去立即数#6400，计算结果影响标志位。

② CMN R2, R1

指令将 R1 寄存器的内容加到 R2 寄存器，计算结果影响标志位。

3.1.10　分支控制指令

分支控制指令(Branch and Control Instructions)包括直接跳转指令 B 和 BL、间接跳转指令 BX 和 BLX，具体指令格式和功能见表 3-10.

表 3-10　跳转指令

示　　例	功 能 描 述
B<cond> <target address>	按<contd>条件决定是否分支
B<tartet address>	无条件分支
BL <Rm>	带链接分支
B{cond}. W <label>	条件分支
BL. W <label>	带链接的分支
BL. W<c> <label>	带链接的分支(立即数)
B. W <label>	无条件分支

- B 是直接跳转指令，label 可以是 24 位的有符号数，跳转范围是－16～16MB。
- BL 除了可以直接跳转到 label 处外，在跳转前会将 PC 内容保存到 LR 寄存器。
- BX 是间接跳转指令，跳转到的目标地址存放在 Rm 寄存器。
- BLX 也是一个间接跳转指令，同样在跳转前会将 PC 内容保存到 LR 寄存器。

使用举例：

```
① B loop          ；直接跳转到 loop 处；
② BLE ng          ；当 Z=1 或 N!=V,即有符号数小于或等于时，直接跳转到 ng 处；
③ BEQ target1     ；当 Z =1,即相等时，直接跳转到 target1 处
④ BX LR           ；返回函数调用处
```

3.1.11　其他指令

主要包括位操作指令、符号扩展指令、字节交换指令等，指令的具体格式和功能见表 3-11。

表 3-11　其他指令

示　　例	功 能 描 述
BFC.W Rd，Rn，#<width>	位区清零
BFI.W Rd，Rn，#<lsb>，#<width>	将一个寄存器的位区插入另一个寄存器中
SBFX.W Rd，Rn，#<lsb>，#<width>	复制位段，并带符号扩展到 32 位
SBFX.W Rd，Rn，#<lsb>，#<width>	复制位段，并无符号扩展到 32 位
REV.W Rd，Rn	在字中反转字节序
REV16.W Rd，Rn	在高低半字中反转字节序
REVSH.W	在低半字中反转字节序，并做带符号扩展
SXTB Rd，Rm{，<rotation>}	Rd ＝ Rm 把带符号字节整数扩展到 32 位
SXTH Rd，Rm{，<rotation>}	Rd ＝ Rm 把带符号半字整数扩展到 32 位

1）BFC 和 BFI

- BFC 指令实现 Rd 寄存器中从 1sb 开始的 width 位数的位清零。
- BFI 指令实现 Rn 寄存器中从 0 开始的 width 位拷贝到 Rd 寄存器中从 1sb 开始的 width 位。

使用举例：

```
① BFC R1, #8, #13
```

该指令实现对 R1 寄存器中从第 8 位开始的 13 位(即到第 20 位)的数据进行清零。

```
② BFI R2, R3, #7, #11
```

该指令实现将 R3 寄存器中从第 0 位到第 10 位的数据拷贝到 R2 寄存器的第 7 位到第 17 位。

2）SBFX 和 UBFX

- SBFX 指令实现将 Rn 寄存器中从第 1sb 位开始的 width 位抽取出来，然后进行有符号位扩展到 32 位并将结果存入 Rd 寄存器。
- UBFX 指令实现将 Rn 寄存器中从第 1sb 位开始的 width 位抽取出来，然后进行无符号位扩展到 32 位并将结果存入 Rd 寄存器。

使用举例：

```
① SBFX R1, R2, #10, #4
```

该指令实现将 R2 寄存器中的第 10 位到第 13 位抽取出来并进行有符号扩展到 32 位，

然后将结果存入 R1 寄存器。

```
② UBFX R3, R4, #9, #10
```

该指令实现将 R4 寄存器中的第 9 位到第 18 位抽取出来并进行无符号扩展到 32 位,然后将结果存入 R3 寄存器。

3) SXTB、SXTH、UXTB 和 UXTH

- SXTB 指令实现将 Rm 寄存器的低 8 位,或 Rm 寄存器经过循环右移 rotation 位后的低 8 位,有符号扩展到 32 位,然后存入 Rd 寄存器。
- UXTH 指令实现将 Rm 寄存器的低 16 位,或 Rm 寄存器经过循环右移 rotation 位后的低 16 位,无符号扩展到 32 位,然后存入 Rd 寄存器。

使用举例:

```
① SXTH R1, R2, ROR #16
```

该指令首先将 R2 寄存器的内容进行循环右移 16 位,然后取出低 16 位的半字并进行有符号扩展到 32 位,最后将结果存入 R1 寄存器。

```
② UXTB R3, R10
```

该指令首先取出 R10 寄存器的低 8 位,然后进行无符号扩展到 32 位,最后将结果存入 R3 寄存器。

4) REV、REV16、REVSH

- REV 实现一个字 Rn 的四个字节大小端转换,复制到 Rd 中。
- REV16 实现 Rn 的两个半字内部的大小端转换,复制到 Rd 中。
- REVSH 将 Rn 低半字内的字节反转,再把反转后的值带符号位扩展到 32 位后,复制到 Rd 中。

字节交换指令的原理见图 3-9。

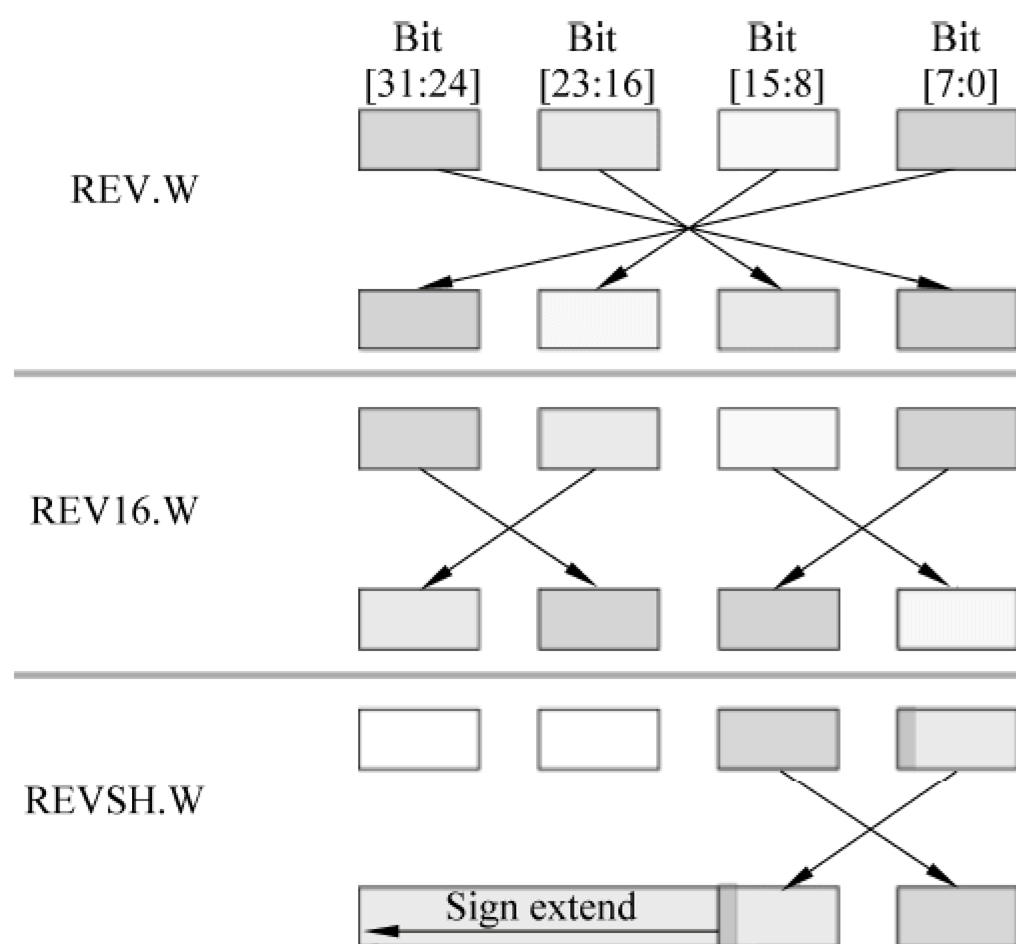


图 3-9 字节交换指令交换顺序

3.2　ARM 汇编器中的伪指令

ARM 汇编语言程序里,有一些特殊指令助记符,这些助记符与指令系统的助记符不同,没有相对应的操作码,通常称这些特殊指令助记符为伪指令。伪指令在源程序中的作用是为完成汇编程序作各种准备工作。有以下几种伪指令:符号定义伪指令、数据定义伪指令、汇编控制伪指令和宏指令。

3.2.1　Thumb 伪指令

1) ADR

小范围的地址读取伪指令。ADR 指令将基于 PC 相对偏移的地址值读取到寄存器中。ADR 伪指令格式如下:

```
ADR register,expr
```

其中,register 为加载的目标寄存器,expr 为地址表达式。偏移量必须是正数并小于 1KB。

ADR 伪指令示例:

```
    ADR R0,TxtTab
    ...
TxtTab:
    DCB "ARM7TDMI",0
```

2) LDR

大范围的地址读取伪指令。LDR 伪指令用于加载 32 位的立即数或一个地址值到指定寄存器。在汇编编译源程序时,LDR 伪指令被编译器替换成一条合适的指令。若加载的常数未超出 MOV 范围,则使用 MOV 或 MVN 指令代替 LDR 伪指令,否则汇编器将常量放入文字池,并使用一条程序相对偏移的 LDR 指令从文字池读出常量。LDR 伪指令格式如下:

```
LDR register,=expr/label_expr
```

其中,register 为加载的目标寄存器,expr 是 32 位立即数,label_expr 是基于 PC 的地址表达式或外部表达式。

LDR 伪指令举例如下:

```
LDR R0,=0x12345678       ;加载 32 位立即数 0x12345678
LDR R0,=DATA_BUF+60      ;加载 DATA_BUF 地址+60
⋮
```

```
LTORG                    ;声明文字池
    ⋮
```

文字池一般由 ARM 编译器自动分配，从 PC 到文字池的偏移量必须是正数小于是 1KB。与 Thumb 指令的 LDR 相比，伪指令的 LDR 的参数有“＝”号。

3）NOP

空操作伪指令。NOP 伪指令在汇编时将会将会被代替成 ARM 中的空操作，比如可能为“MOV R8，R8”指令等，可用于延时操作。NOP 伪指令格式如下：

```
NOP
```

3.2.2 符号定义伪指令

符号定义伪指令用于定义汇编程序中的变量、进行变量赋值以及定义寄存器的别名等操作。常见的符号定义伪指令包括：

1）定义全局变量的伪指令 GBLA、GBLL 和 GBLS

语法格式：

```
GBLA(GBLL 或 GBLS) 全局变量名
```

GBLA、GBLL 和 GBLS 伪指令用于定义一个 ARM 程序中的全局变量，并将其初始化。其中：

- GBLA 伪指令用于定义一个全局的数字变量，并初始化为 0；
- GBLL 伪指令用于定义一个全局的逻辑变量，并初始化为 F(假)；
- GBLS 伪指令用于定义一个全局的字符串变量，并初始化为空。

使用示例：

```
GBLA T1                  ;定义一个全局的数字变量,变量名为 T1,初始值为 0
GBLL T2                  ;定义一个全局的逻辑变量,变量名为 T2,初始值为 F
GBLS T3                  ;定义一个全局的字符串变量,变量名为 T3,初始值为空
```

2）定义局部变量的伪指令 LCLA、LCLL 和 LCLS

语法格式：

```
LCLA(LCLL 或 LCLS) 局部变量名
```

LCLA、LCLL 和 LCLS 伪指令用于定义一个 ARM 程序中的局部变量，并将其初始化。其中：

- LCLA 伪指令用于定义一个局部的数字变量，并初始化为 0；
- LCLL 伪指令用于定义一个局部的逻辑变量，并初始化为 F(假)；
- LCLS 伪指令用于定义一个局部的字符串变量，并初始化为空。

定义局部变量的伪指令的使用方法与定义全局变量的伪指令的使用方法类似。

3）进行变量赋值的伪指令 SETA、SETL 和 SETS

语法格式：

```
变量名 SETA(SETL或 SETS) 表达式
```

伪指令 SETA、SETL、SETS 用于给一个已经定义的全局变量或局部变量赋值。其中：

- SETA 伪指令用于给一个数学变量赋值；
- SETL 伪指令用于给一个逻辑变量赋值；
- SETS 伪指令用于给一个字符串变量赋值。

使用示例：

```
LCLL T4             ;定义一个局部的逻辑变量,变量名为 T4
T4 SETL {TRUE}      ;将该逻辑变量赋值为 TRUE
```

3.2.3 数据定义伪指令

数据定义伪指令一般用于为特定的数据分配存储单元并进行初始化。常见的数据定义伪指令包括：

1）连续分配一片连续的字节存储单元的伪指令 DCB

语法格式：

```
标号 DCB表达式
```

DCB 伪指令用于分配一片连续的字节存储单元并用伪指令中指定的表达式初始化。其中，表达式可以为 0～255 的数字或字符串。

使用示例：

```
MyName DCB "This is my name."
```

分配一片连续的字节存储单元并初始化，起始地址为 MyName。

2）连续分配一片连续的半字存储单元的伪指令 DCW（或 DCWU）

语法格式：

```
标号 DCW(或 DCWU) 表达式
```

DCW（或 DCWU）伪指令用于分配一片连续的半字存储单元并用伪指令中指定的表达式初始化。使用 DCW 分配的字存储单元是半字对齐的，而用 DCWU 分配的字存储单元并不严格半字对齐。

使用示例：

```
WTest DCW 1,2,3;
```

分配 3 个连续的半字存储单元并初始化为 1，2，3，起始地址为 Wtest。

3）连续分配一片连续的字存储单元的伪指令 DCD（或 DCDU）

语法格式：

```
标号 DCD(或 DCDU) 表达式
```

DCD（或 DCDU）伪指令用于分配一片连续的字存储单元并用伪指令中指定的表达式初始化。用 DCD 分配的字存储单元是字对齐的，而用 DCDU 分配的字存储单元并不严格字对齐。

使用示例：

```
DTest DCD 4,5,6 ;
```

分配 3 个连续的字存储单元并初始化为 4，5，6，起始地址为 Dtest。

4）分配一片连续的存储单元的伪指令 SPACE

语法格式：

```
标号 SPACE 表达式
```

SPACE 伪指令用于分配一片连续的存储区域并初始化为 0。其中，表达式为要分配的字节数。

使用示例：

```
DataSpace SPACE 10;
```

分配连续 10 个字节的存储单元并初始化为 0，起始地址为 DataSpace。

3.2.4 汇编控制伪指令

汇编控制伪指令用于控制汇编程序的执行流程，常用的汇编控制伪指令包括如下两条：

1）IF、ELSE、ENDIF

语法格式：

```
IF 逻辑表达式
指令序列 1
ELSE
指令序列 2
ENDIF
```

IF、ELSE、ENDIF 伪指令能根据条件的成立与否决定是否执行某个指令序列。当 IF 后面的逻辑表达式为真，则执行指令序列 1，否则执行指令序列 2。其中，ELSE 及指令序列 2 可以没有，此时，当 IF 后面的逻辑表达式为真，则执行指令序列 1，否则继续执行后面的指令。

使用示例：

```
GBLL Flag
```

```
…
IF Flag = TRUE
指令序列 1
ELSE
指令序列 2
ENDIF
```

2）WHILE、WEND

语法格式：

```
WHILE 逻辑表达式
指令序列
WEND
```

WHILE、WEND 伪指令能根据条件的成立与否决定是否循环执行某个指令序列。当 WHILE 后面的逻辑表达式为真，则执行指令序列，该指令序列执行完毕后，再判断逻辑表达式的值，若为真则继续执行，一直到逻辑表达式的值为假。

使用示例：

```
GBLA Counter
Counter SETA 3
⋮
WHILE Counter < 10
指令序列
WEND
```

3.2.5 其他常用的伪指令

其他的伪指令包括：

1）定义代码段或数据段伪指令 AREA

AREA 伪指令用于定义一个代码段或数据段，其语法格式如下。

```
AREA 段名 属性 1,属性 2,…
```

使用示例：

```
AREA Init,CODE,READONLY
```

该伪指令定义了一个代码段，段名为 Init，属性为只读。

2）指令标识伪指令 CODE16、CODE32

CODE16 伪指令通知编译器，其后的指令序列为 16 位的 Thumb 指令。

CODE32 伪指令通知编译器，其后的指令序列为 32 位的 ARM 指令。

使用示例：

```
AREA Init,CODE,READONLY
…
CODE32              ;通知编译器其后的指令为32位的ARM指令
CODE16              ;通知编译器其后的指令为16位的Thumb指令
…
```

3）指定应用程序入口的伪指令ENTRY

ENTRY伪指令用于指定汇编程序的入口点。

使用示例：

```
AREA Init,CODE,READONLY
ENTRY               ;指定应用程序的入口点
…
```

4）指定应用程序结束的伪指令END

END伪指令用于通知编译器已经到了源程序的结尾。

使用示例：

```
AREA Init,CODE,READONLY
…
END                 ;指定应用程序的结尾
```

3.3 汇编语言的程序结构

汇编语言程序中，以程序段为单位组织代码。段是相对独立的指令或数据序列，具有特定的名称。段可以分为代码段和数据段，代码段的内容为执行代码，数据段存放代码运行时需要用到的数据。一个汇编程序至少应该有一个代码段，多个段在程序编译链接时最终形成一个可执行文件。

以下是一个汇编语言源程序的基本结构：

```
AREA Init,CODE,READONLY
ENTRY
  LDR R0,=0x3FF5000
  LDR R1,0xFF
  STR R1,[R0]
  LDR R0,=0x3FF5008
  LDR R1,0x01
  STR R1,[R0]
END
```

本例定义了一个名为Init的代码段，属性为只读。ENTRY伪指令标识程序的入口点，接下来为指令序列，程序的末尾为END伪指令，该伪指令告诉编译器源文件的结束，每一

个汇编程序段都必须有一条 END 伪指令，指示代码段的结束。多加几个例子说明汇编程序设计。

1）循环累加案例

用汇编实现 1＋2＋3＋…＋10 的累加计算，代码如下。

```
STACK_TOP EQU 0x2000 0200                    ;栈地址
AREA Init,CODE,READONLY
ENTRY
DCD STACK_TOP                                ;复位后建立栈指针
DCD START                                    ;复位后执行的代码地址
START:
    MOVS R0, #10                             ;初始化
    MOVS R1, #0
;计算 10+9…+1
LOOP:
    ADDS R1, R0                              ;R1=R1+R0
    SUBS R0, #1                              ;R0=R0-1
    BNE LOOP                                 ;不为 0 跳转
    LDR R0, =RESULT                          ;获取数据存储区地址
    STR R1, [R0]                             ;结果现在存储到 R1 中了
DEADLOOP:
    B DEADLOOP
;数据区定义
AREA BUF,DATA,READWRITE
RESULT:
    DCD 0                                    ;数据存储区
    END
```

2）启动代码分析

启动代码是处理器上电后执行的第一部分代码，启动代码完成程序的栈空间、堆空间以及复位中断和其他中断的中断向量入口设置，最后跳转到 Main 函数执行。根据不同的处理器，CMSIS 库提供不同的启动代码。以下对 STM32L152 系列处理器的启动代码进行分析。

启动代码中首先定义了栈、堆的大小。

```
Stack_Size      EQU     0x00000400           ;栈大小
AREA    STACK, NOINIT, READWRITE, ALIGN=3    ;定义数据段,8字节对齐
Stack_Mem       SPACE   Stack_Size           ;开辟栈空间
__initial_sp                                 ;栈顶标号

Heap_Size       EQU     0x00000200           ;堆大小
AREA    HEAP, NOINIT, READWRITE, ALIGN=3     ;定义数据段
```

```
__heap_base                                  ;堆的起始地址
Heap_Mem        SPACE   Heap_Size            ;开辟堆空间
__heap_limit                                 ;堆的结束地址

PRESERVE8                                    ;指示编译器 8 字节对齐
THUMB                                        ;指示编译器为 THUMB 指令
```

其次,启动程序中定义了所有的中断向量名称及其入口地址,中断向量表在复位时映射到 0 地址。

```
AREA     RESET, DATA, READONLY               ;定义代码段,位于 0 地址
;申明三个标号
EXPORT   __Vectors
EXPORT   __Vectors_End
EXPORT   __Vectors_Size
;用 DCD 定义一个字存储空间,存放后面符号的地址,这些符号名称在 stm32L1xx_it.c 中是一个函
;数名,在该函数中添加代码即可实现中断处理
__Vectors       DCD     __initial_sp         ; 栈顶地址
                DCD     Reset_Handler        ; 复位中断函数
                DCD     NMI_Handler          ; NMI 中断函数
                DCD     HardFault_Handler    ; Hard Fault Handler
                DCD     MemManage_Handler    ; MPU Fault Handler
                DCD     BusFault_Handler     ; Bus Fault Handler
                DCD     UsageFault_Handler   ; Usage Fault Handler
                ...
                DCD     TIM6_IRQHandler      ; TIM6
                DCD     TIM7_IRQHandler      ; TIM7
__Vectors_End                   ;标记中断向量表的结束地址
__Vectors_Size  EQU __Vectors_End-__Vectors  ;计算中断向量表大小
```

启动代码的主程序部分是复位中断处理函数。

```
AREA     |.text|, CODE, READONLY             ;定义代码段
Reset_Handler     PROC                       ;定义复位函数
EXPORT   Reset_Handler       [WEAK]
IMPORT   __main                              ;导入__main 函数的地址
IMPORT   SystemInit                          ;导入 SystemInit 函数的地址
         LDR     R0, =SystemInit
         BLX     R0                          ;跳转到 SystemInit 运行
         LDR     R0, =__main
         BX      R0                          ;跳转到__main 函数执行
ENDP
```

其中 EXPORT Reset_Handler [WEAK]是导出 Reset_Handler 标识,[WEAK]用来

表明如果由外部定义的其他 Reset_Handler 函数，则执行其他 Reset_Handler 函数。

SystemInit 函数在 system_stm32l1xx.c 中，其功能是初始化 flash 接口，设置启动时钟。

__main 是系统提供的主程序调用库，__main 函数主要执行的功能包括：

- 加载 RO 和 RW 代码及数据；
- 初始化静态存储区数据为 0；
- 初始化堆栈；
- 跳转到 C 语言的 main 函数执行；
- 处理 main 函数的返回值。

【思考题：__main 和 C 语言的 main 函数是一个函数吗？**】**

Chapter 4

第 4 章　开发板硬件系统及开发环境

【导读】 本章为嵌入式系统开发的基础知识，首先介绍嵌入式硬件最小系统的概念的组成，典型的外围电路原理，然后对嵌入式开发流程、CMSIS 库的结构和功能进行详细阐述。针对嵌入式程序设计中涉及的 C 语言常用知识，如宏定义、Volatile、位与、位或、按位取反、左移、右移、寄存器操作等基础知识，本章进行了简要梳理。本章的目的为建立嵌入式开发的流程，掌握嵌入式 C 开发的基础知识。

4.1　最小系统设计

一个嵌入式处理器自己不能独立工作，必须要加上电源、提供复位和时钟信号，如果嵌入式处理器片内没有存储器，还需要加上片外 Flash、RAM 构成一个系统才能正常工作。嵌入式处理器运行所必需的电路和嵌入式处理器构成了嵌入式处理器的最小系统。系统的调试接口在运行时不是必需的，但在开发时必须要使用，因此调试接口也是最小系统组成之一。

最小系统的基本组成如图 4-1 所示。

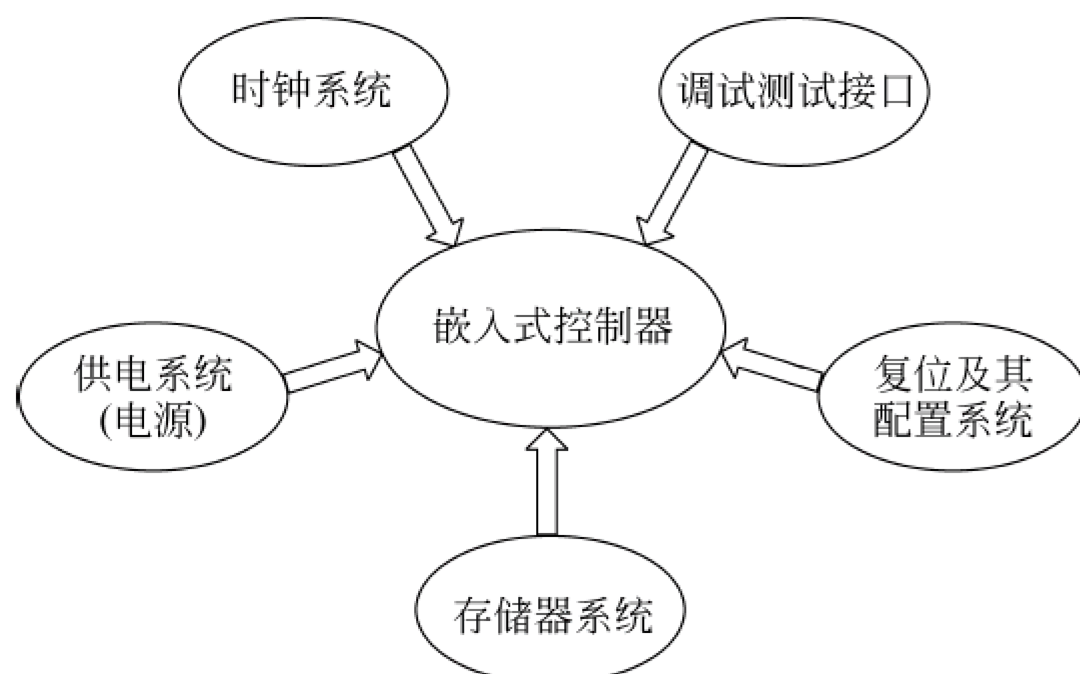


图 4-1　嵌入式系统的最小组成

(1) **供电系统**：电源系统为整个系统提供能量，是系统工作的基础。嵌入式处理器的电源一般采用 5V、3.3V、1.8V 等直流供电，分为数字电源和模拟电源，模拟电源一般用于 AD 采集模块，两个电源在要求不高的情况下可以不分开。为保证处理器供电的稳定性，一

般采用电源芯片对输入电压进行稳压，在设计时需要考虑到电源的功率是否满足系统需求。

(2) **存储系统**：对于大多数嵌入式处理器，其内部都带有片内 Flash 和 RAM，这样外部无需增加存储器，如果内部不带存储器或者容量不能满足系统需求，则需要外扩存储器，一般通过外部总线进行连接。

(3) **复位电路**：嵌入式处理器在上电工作时，需要将处理器的状态和内部寄存器初始化为一个确认的状态，以防止程序运行不能正确执行，因此需要给处理器一个复位信号。复位信号一般要求持续一定时间，在上电时可以通过阻容复位电路给出，也可以通过专用的复位芯片给出。在系统工作过程中，如果碰到电源电压过低、干扰等可能导致嵌入式处理器无法正常工作的情况，复位芯片给出复位信号。

(4) **调试接口**：嵌入式处理器一般带有 JTAG 调试接口，通过 JTAG 调试接口可以控制芯片的运行并获取内部信息。ARM Cortex 系列处理器采用了新的 CoreSight，内核本身不再含有 JTAG 接口。取而代之的是调试访问接口(DAP)，由一个在芯片内部实现的调试端口设备(DP)完成，也支持 JTAG 调试方法。

(5) **时钟系统**：嵌入式处理器一般为时序电路，需要时钟信号才能工作，大多数嵌入式处理器内部集成振荡器，作为时钟源；内部时钟一般精确度较低，因此在一些时序要求严格的场合需要外部晶振，此外，对于低功耗应用，嵌入式处理器在很低的功耗下需要外部时钟唤醒，此时需要在电路上增加外部晶振。

4.2　开发板电路原理图

本教材使用的开发板为 ST 的 STM32L-Discovery 开发板，开发板集成一个 STLINK 调试器、两个按键、一个显示屏、一个触摸按键(可作为 4 个按键)，并将所有的芯片输入输出引脚引出便于扩展使用。开发板的结构如图 4-2 所示。以下将分别对开发板电路各个部分的原理进行介绍。

4.2.1　电源

STM32L-Discovery 开发板提供 5V 和 3.3V 两个电源，开发板电源可采用 PC USB 口供电，或者通过单独的 5V 或 3.3V 直流电源供电。

如图 4-3 所示，5V 的直流电源经过一个稳压芯片 LD3985M33 后转换为 3.3V，开发板在扩展引脚 EXT_5V 和 EXT_3V 上将 5V 和 3.3V 的两个电源单独引出，可以作为输出电源供给其他电路(电流小于 100mA)。图 4-3 中的二极管 D1 和 D2 对 3.3V 电源和 5V 电源进行了保护，使得开发板也可以在扩展引脚 EXT_5V 和 EXT_3V 上外接 5V 和 3.3V 电源作为输入电源。

为便于电池供电的低功耗应用设计，开发板提供了一个纽扣电池供电电路，如图 4-4 所示可通过板子的跳线进行电源切换。

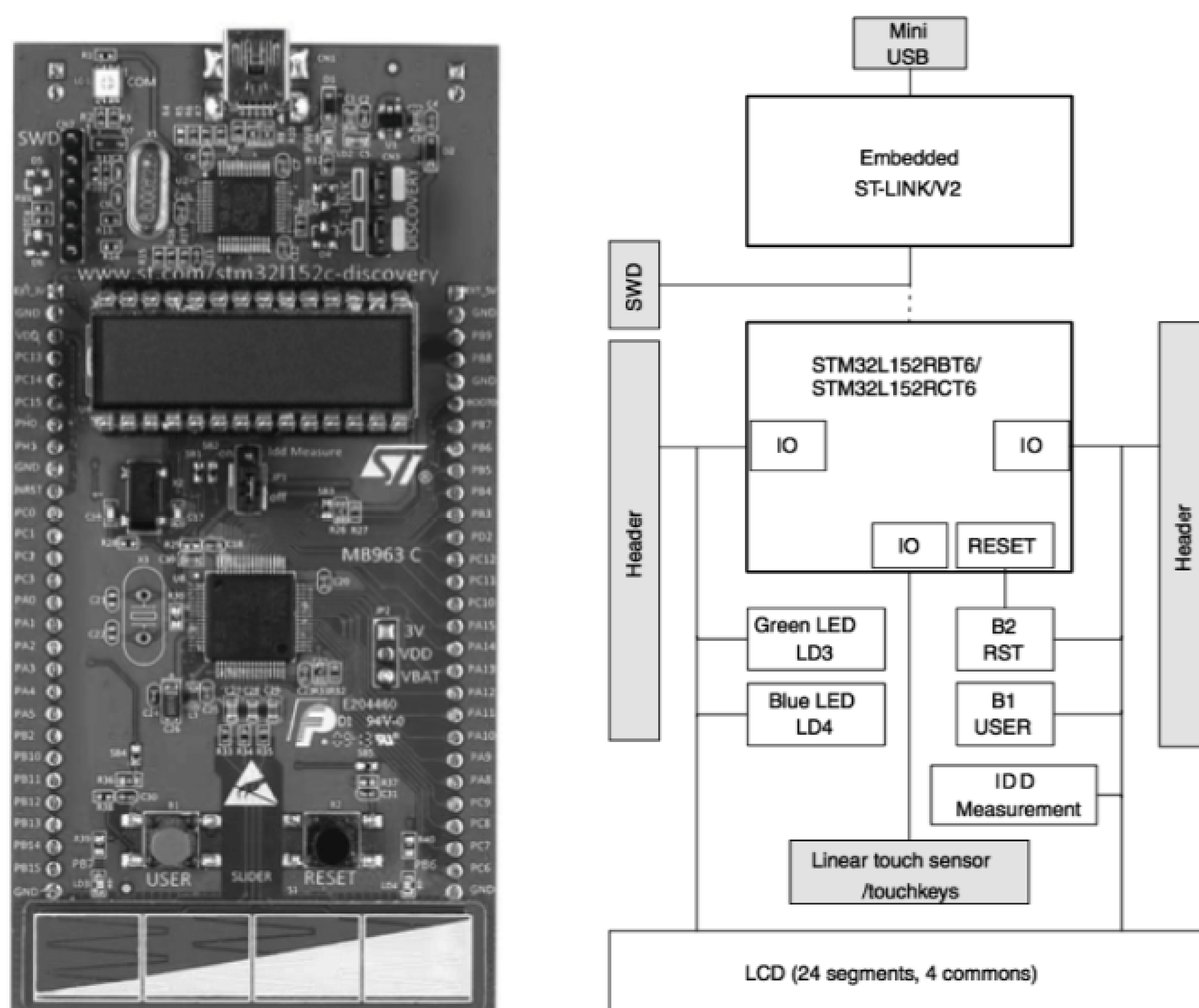


图 4-2 STM32L152-Discovery 开发板

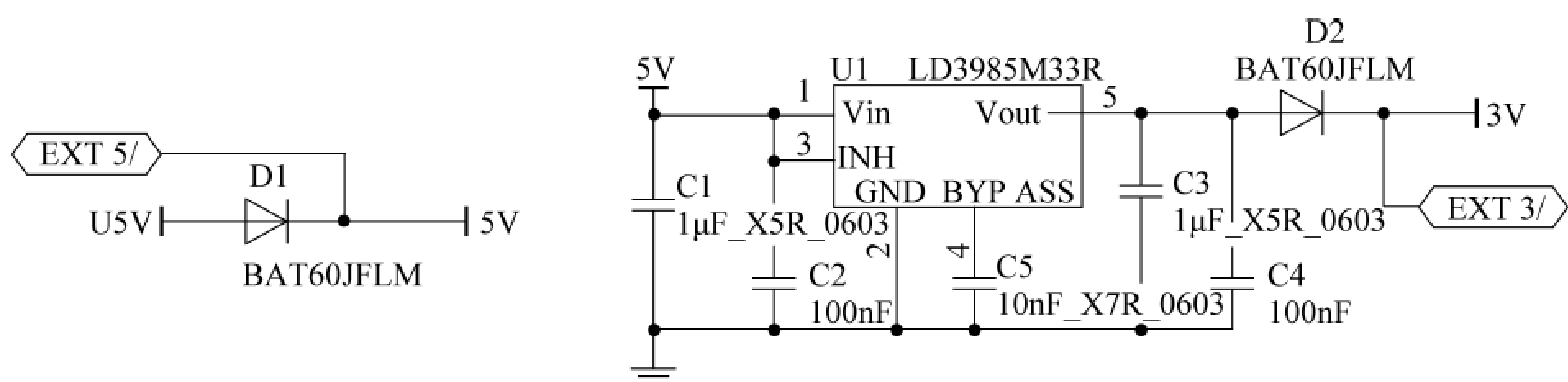


图 4-3 电源电路原理

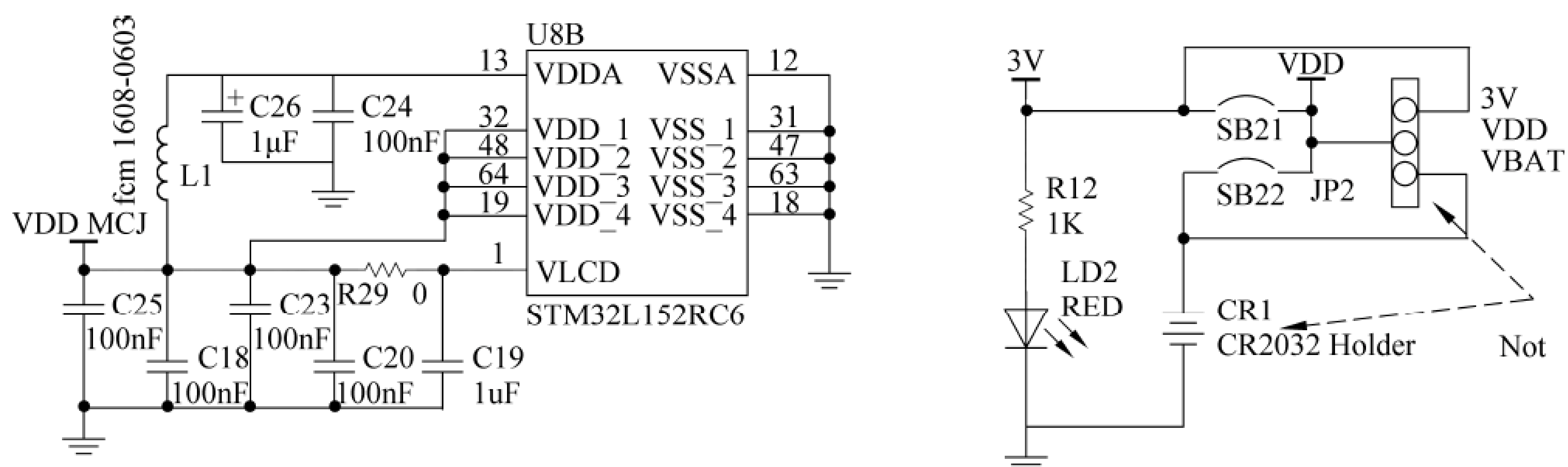


图 4-4 电池切换电路

STM32L152 处理器的电源包括模拟电源 VDDA，数字电源 VDD 和液晶屏电源 VLCD，供电电压为 3.3V。模拟电源用于内部 AD 转换器，LCD 电源供给液晶屏，数字电源为其他控制器和 I/O 使用。在电源输入端进行滤波处理，保障电源的稳定性。

4.2.2 复位和启动电路

STM32L152 采用低电平复位，STM32L-Discovery 提供了一个阻容复位电路，用户可通过复位按键实现手动复位。上电时，复位引脚 NRST 为低，电容 C31 充电，充满后 NRST 被置高，完成复位。当用户按下复位键时，电容 C31 放电，NRST 被拉低，按键抬起后，电容充电，NRST 被拉高，完成复位。STM32L152 含内部复位电路，当 VDD 引脚电压小于 1.65V 时，芯片会保持在复位状态，如图 4-5 所示。

STM32L152 复位后开始执行程序，执行程序的位置与芯片的启动引脚 Boot0 和 Boot1 的状态有关，一般情况下，配置成 Flash 启动，即 Boot0 为低电平，复位后处理器从用户程序开始执行。在通过 ISP 下载程序的模式下，需要先运行片内 Flash 的 Bootloader 程序，该程序可以通过串口、USB 口、SPI、IIC 等方式将用户程序写入到用户 Flash，此时置 Boot1 为低电平，Boot0 为高电平。Boot1 和 Boot0 都为高电平时，复位后从 RAM 执行，由于掉电后 RAM 数据不保存，此模式一般只作为调试时使用，如图 4-6 所示。

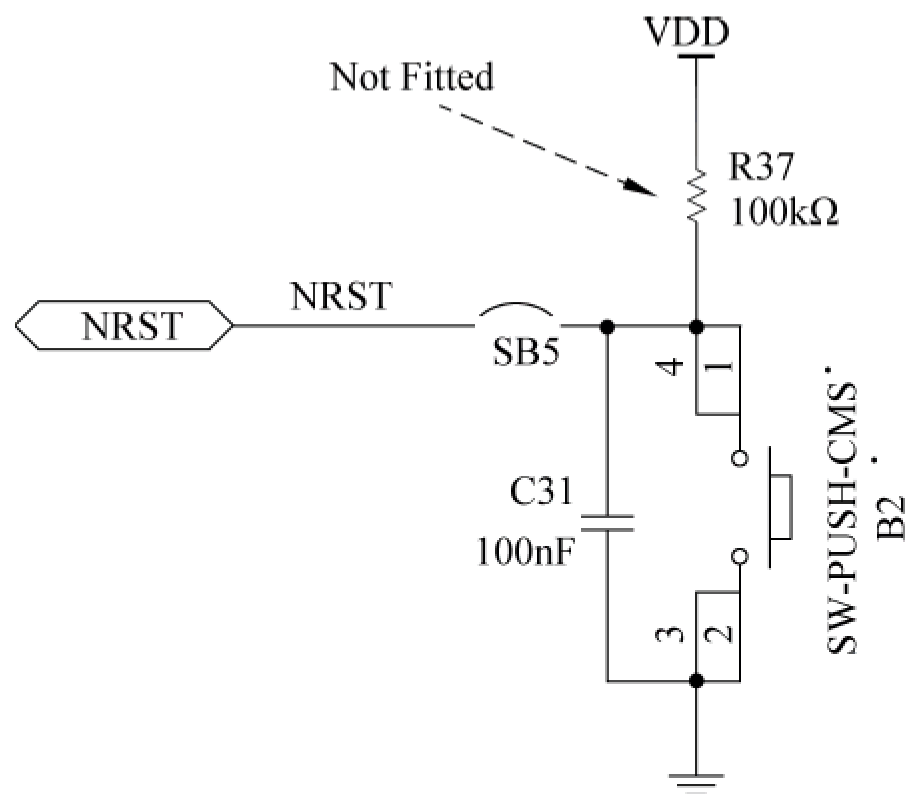


图 4-5 复位电路

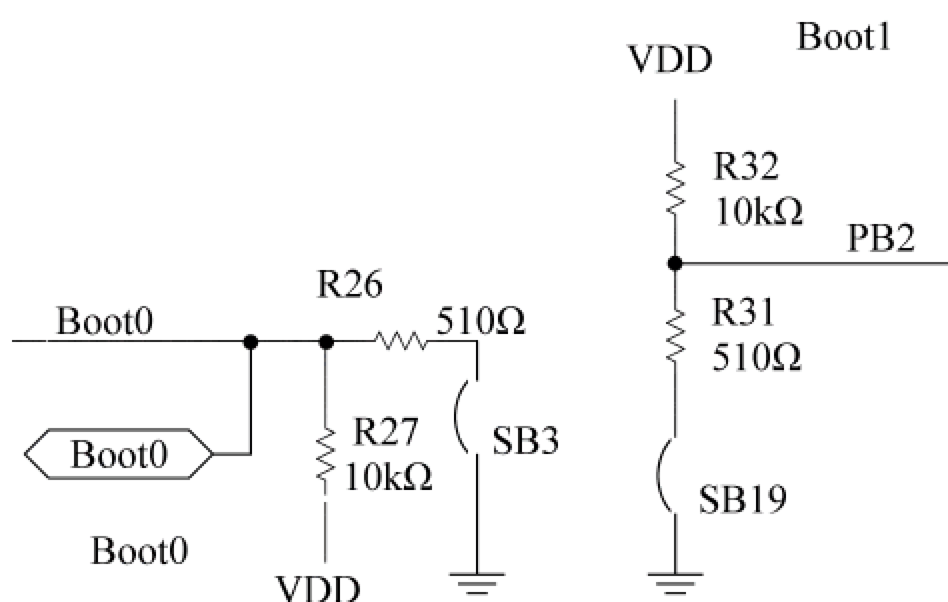


图 4-6 启动选择电路

STM32L-Discovery 开发板的 Boot0 默认置高，Boot1 由 PB2 引脚控制，电路板上电后，先执行 MCU 内部 Bootloader 程序，再跳转到用户代码去执行。Boot0 和 Boot1 的电平状态可以通过开发板硬件配置开关 SB3 和 SB19 更改。

4.2.3 时钟

STM32L152 支持多种时钟源，包括高速外部振荡器 HSE、低速外部振荡器 LSE、高速内部振荡器 HIS、多速率内部振荡器和低速内部振荡器 LSI。STM32L152 带有 16MHz 内

部 RC 振荡器 HSI，7 种频率的多速率内部 RC 振荡器 MSI，可以为 PLL 锁相环提供时钟，PLL 将时钟倍频到 32MHz 满速运行。37kHz 的低速 RC 振荡器 LSI 可用于实时时钟 RTC 以及看门狗 WDG。但由于内部 RC 振荡不精确，起振不稳定，因此一般会在开发板外置晶振。

外置晶振一般有两种：一种是高速外部晶振 HSE，可用作处理器的主时钟和外围控制器时钟，工作频率为 1M～24MHz；另一种是低速外部晶振 LSE，工作频率 32.768kHz，主要用于驱动实时时钟 RTC 和看门狗 WDG。外部晶振精度较高，而且频率越低精度越容易控制，因此一般 Timer，RTC 用 32.768kHz 的低速外部晶振。图 4-7 和图 4-8 是 STM32L-Discovery 开发板的外部时钟电路，高速外部晶振没有焊接，系统默认使用内部 RC 振荡器作为主时钟。



图 4-7　外部低速晶振电路



图 4-8　外部高速晶振电路

4.2.4　调试接口

STM32L152 采用 CoreSight 调试系统，内部没有 JTAG，而是调试访问接口 DAP，DAP 可以以 SWD 方式或 JTAG 的方式对外提供调试功能。SWD 只需要最少 2 根线

(SWCLK 和 SWDIO),JTAG 需要使用 5 根线。JTAG 接口和 SWD 接口共用,因此通过 JTAG 接口也可以采用 SWD 方式下载调试。STM32L-Discovery 板载一个 STLINK 调试器,该调试器与 STM32L152 处理器通过 SWD 方式连接,同时通过板子上的跳线可以单独使用该调试器连接其他处理器进行调试。图 4-9 为典型的 JTAG 调试接口电路,主要包含测试系统复位信号 nTRST、测试数据串行输入 TDI、测试模式选择 TMS、测试时钟 TCK 和测试数据串行输出 TDO,一般 JTAG 调试器为 20 口,将标准 JTAG 的 5 针进行了扩展。SWD 只使用 JTAG 中的 TCK 和 TMS,分别对应 SWCLK 和 SWDIO。目前大量 Cortex 系列处理器的调试器都支持 SWD 模式且占用的 I/O 口和面积小,因此推荐使用 SWD 模式。

4.2.5　按键

按键是系统设计中的主要输入源,STM32L-Discovery 开发板共有两个按键,其中一个用于系统复位,另一个为用户使用,其原理图如图 4-10 所示。按键连接在 PA0 口上,当按键按下时,PA0 为低电平,弹起时为高电平。该按键也可以用作 STM32L152 处理器唤醒输入或者外部中断输入。此外,STM32L-Discovery 开发板的触摸输入也可作为 4 个按键使用。

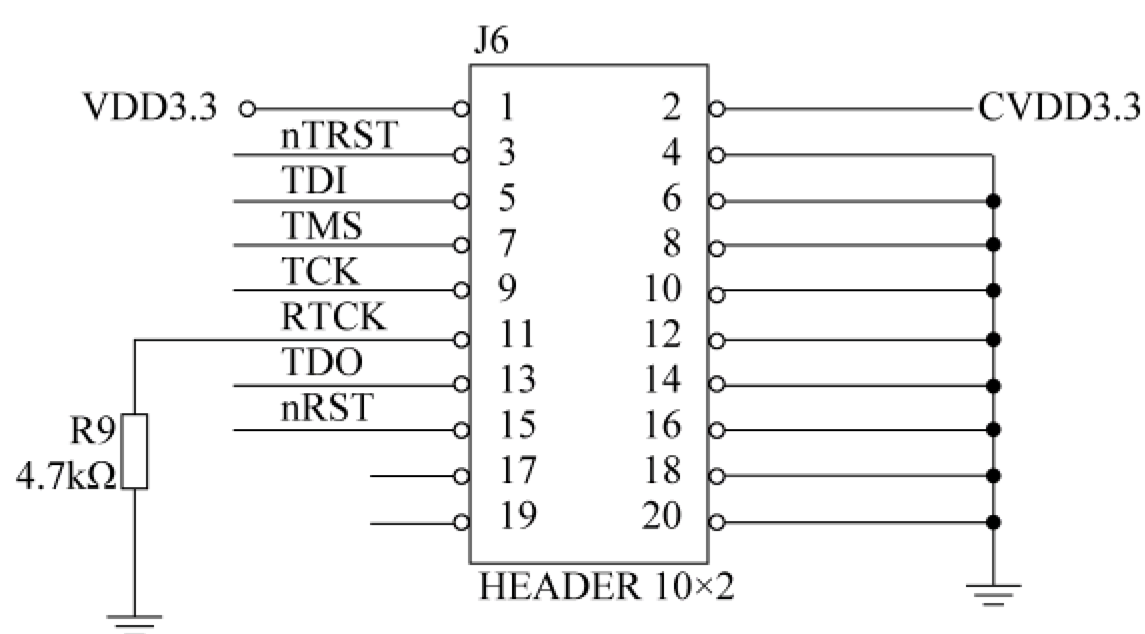


图 4-9　JTAG 调试接口电路

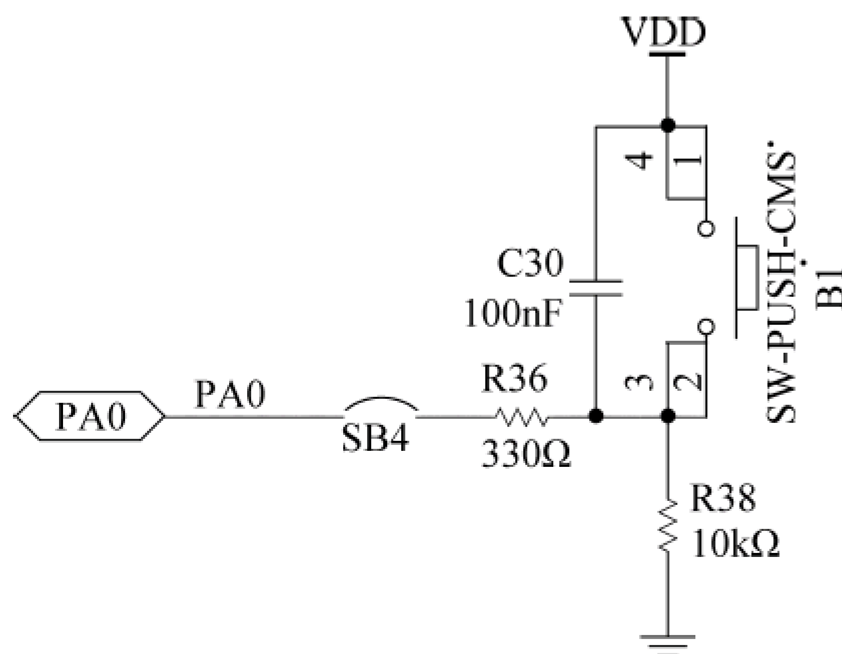


图 4-10　按键电路

4.2.6　LED 灯

STM32L-Discovery 开发板有 2 个用户 LED 灯,原理图如图 4-11 所示,LD3 和 LD4 分别连接在 PB7 和 PB6 上,当 PB7 输出为高电平时 LED3 变亮,反之变灭。此外,开发板提供了 2 个指示灯,其中 LD1 变绿色时指示 STLINK 调试器和处理器之间正在通信;LD2 为电源灯。

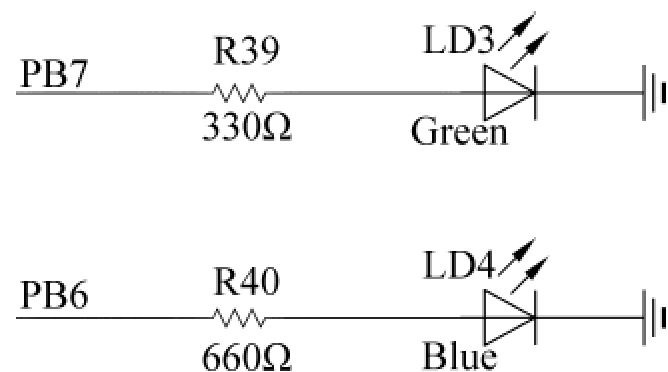


图 4-11　LED 灯电路

4.2.7 显示屏

液晶显示器(Liquid Crystal Display, LCD)是嵌入式系统常用的显示器件,由像素点或符号段组成,每个像素点或符号段是在两电极间放置液态的晶体,当电极间电压大于阈值电压时,控制杆状水晶分子改变方向,将光线折射出来变成可见。电极间的电压要交替变化以保护 LCD 不被损坏。

LCD 控制器的功能是产生显示驱动信号,驱动 LCD 显示器,用户只需要读写一系列的寄存器,完成配置和显示控制。STM32L152 内部集成 LCD 控制器,可以外接无源驱动的单色被动式 LCD 屏幕,最多可接 8 个 Common 端和 44 个 Segment 端的 320 像素 LCD。Common 端为水平方向,Segment 端为竖直方向,两者交叉即可控制像素的显示。

4.2.8 扩展 I/O 口

为便于扩展连接其他外设,STM32L152-Discovery 开发板将 CPU 的 I/O 引脚和电源连接到扩展插针上,如图 4-12 所示。其中包括 3V 和 5V 电源,以及除 PB0,PB1,PA6,PA7,PC4,PC5 之外的所有 I/O 引脚。

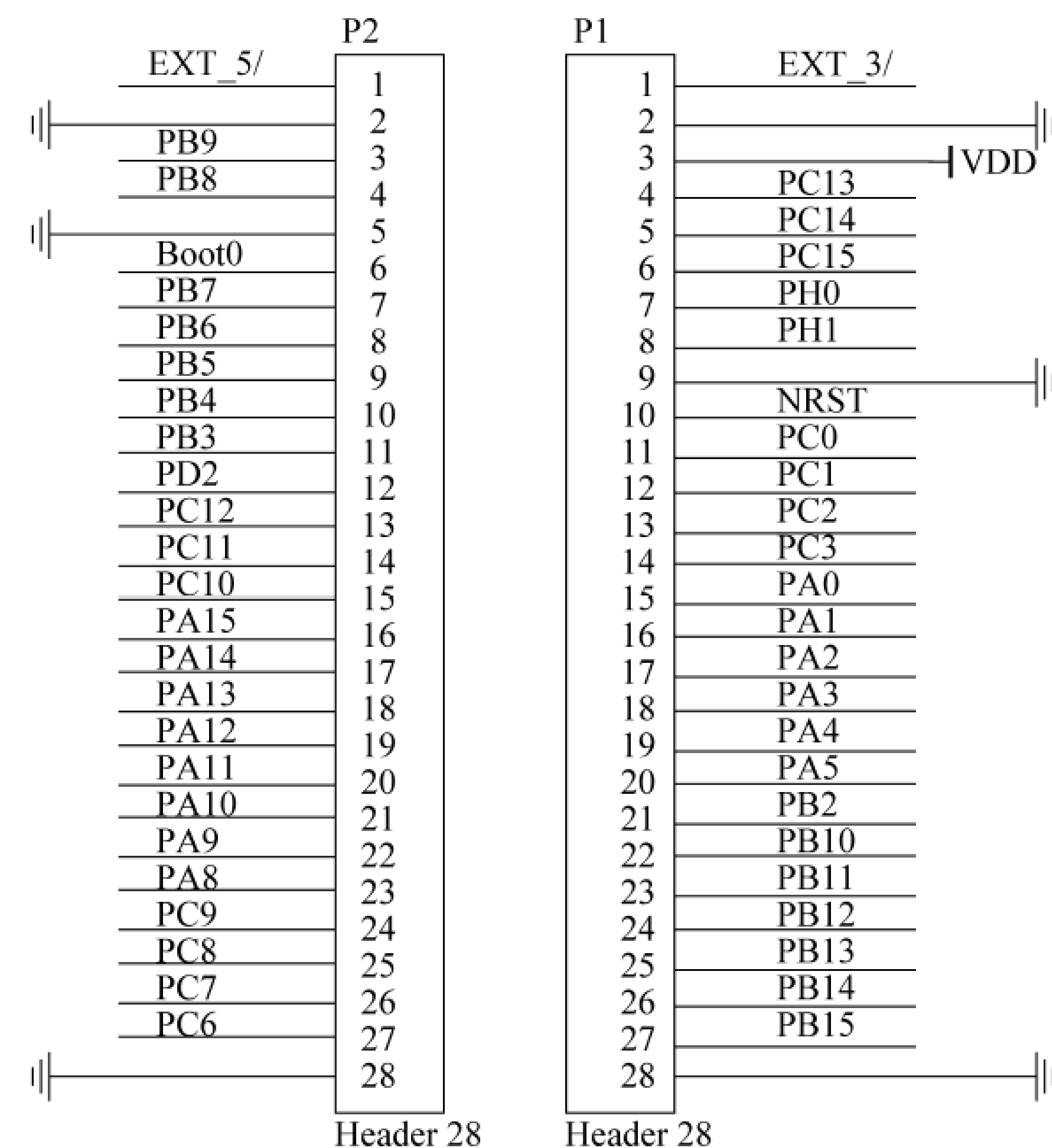


图 4-12 STM32L152-Discovery 扩展接口

扩展 I/O 引脚中,Boot0 已被置为高电平,NRST 为复位引脚,PA0 为按键和休眠唤醒

输入引脚，PA4 为电流采集输入引脚，PC13 为电流采集控制引脚，PA13，PA14，PB3 为 SWD 调试接口，PB6，PB7 为 LED 灯接口，PC14，PC15，PH0，PH1 分别为两个晶振引脚。其余的 33 个扩展 I/O 引脚中，除 PA5，PA11，PA12，PC12，PD2 外，被 LCD 占用。因此外接其他设备时，若需要使用特定功能的引脚或者多于 5 个普通 I/O 引脚，需要去除板子的 LCD 模块。

4.3　软件开发环境

4.3.1　嵌入式软件开发流程

嵌入式系统的软件开发与通用系统的软件开发有较大的区别，以下从交叉编译、交叉调试和固件(firmware)下载对嵌入式软件的编译、调试和固化进行介绍。

1. 交叉编译

程序首先要通过编译器将其转化为 CPU 可执行的机器代码，由于不同的处理器指令集不同，其所需的编译器也不同。在 PC 上开发程序，其编译的程序代码生成的是该 PC 的机器代码，也称为本地编译。

嵌入式软件开发采用交叉编译。交叉编译是指在一个平台上生成可以在另一个平台上可执行的代码的过程。由于目标嵌入式平台资源有限，无法或不便于进行程序的编辑、编译，因此我们在 PC 平台对目标嵌入式平台的程序进行编译，生成目标嵌入式平台的可执行代码。交叉编译的连接示例如图 4-13 所示。

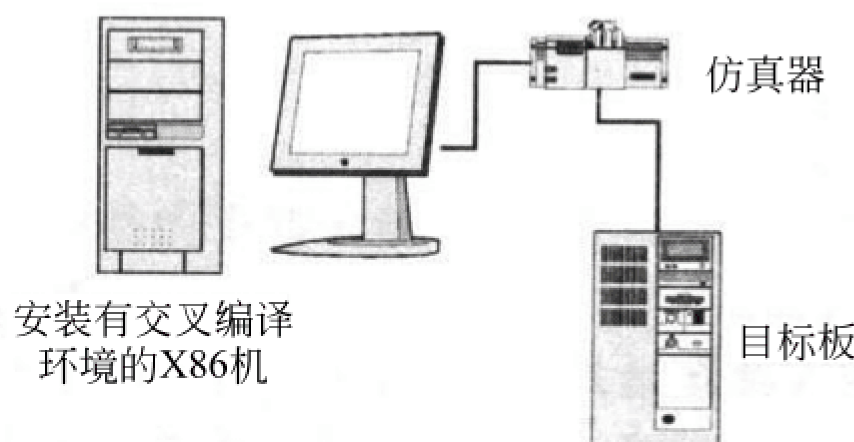


图 4-13　交叉编译环境

由于编译的过程包括编译和链接，因此，嵌入式的交叉编译也包括交叉编译、交叉链接，通常 ARM 的交叉编译器为 arm-elf-gcc、arm-linux-gcc 等，交叉链接器为 arm-elf-ld、arm-linux-ld 等。

2. 交叉调试

软件调试是软件开发过程中必不可少的一个环节，嵌入式软件的交叉调试与通用软件的调试方式有很大的差别。软件开发中，调试器与被调试的程序往往运行在同一台计算机上，调试器是一个单独运行着的进程，它通过操作系统提供的调试接口来控制被调试的进

程,实现单步、断点、变量查看等功能。而在嵌入式软件开发中,调试时采用的是在宿主机和目标机之间进行的交叉调试,调试器仍然运行在宿主机的操作系统之上,但被调试的程序却是运行在基于特定硬件平台的嵌入式操作系统中,调试器和被调试程序通过串口、网络或特殊硬件进行通信,调试器可以控制、访问被调试程序,读取或改变被调试程序的当前状态。

嵌入式系统的交叉调试主要分为软件方式和硬件方式两种。

1) 软件方式

软件调试主要是通过插入调试桩的方式来进行的。调试桩方式进行调试是通过目标操作系统和调试器内分别加入某些功能模块,二者互通信息来进行调试。该方式的典型调试器有 gdb 调试器,一般只能用于调试运行于目标操作系统之上的应用程序,而不宜用来调试目标操作系统的内核代码及启动代码。

2) 硬件调试

相对于软件调试而言,使用硬件调试器可以获得更强大的调试功能和更优秀的调试性能。硬件调试器的基本原理是通过仿真硬件的执行过程,让开发者在调试时可以随时了解到系统的当前执行情况。目前嵌入式系统开发中最常用到的硬件调试器有两种。

In-Circuit Emulator(ICE)方式:ICE 进行交叉调试时需要使用在线仿真器,它是目前最为有效的嵌入式系统的调试手段。它是仿照目标机上的 CPU 而专门设计的硬件,可以完全仿真处理器芯片的行为。仿真器与目标板可以通过仿真头连接,与宿主机可以通过串口、并口、网线或 USB 口等连接方式。由于仿真器自成体系,所以调试时既可以连接目标板,也可以不连接目标板。

在线仿真器提供了非常丰富的调试功能。在使用在线仿真器进行调试的过程中,可以按顺序单步执行,也可以倒退执行,还可以实时查看所有需要的数据,从而给调试过程带来了很多的便利。嵌入式系统应用的一个显著特点是与现实世界中的硬件直接相关,并存在各种异变和事先未知的变化,从而给微处理器的指令执行带来各种不确定因素,这种不确定性在目前情况下只有通过在线仿真器才有可能发现,但其价格比较昂贵,且不同的处理器需要不同的 ICE 硬件。

In-Circuit Debugger(ICD)方式:ICD 交叉调试时需要使用在线调试器,CPU 直接在其内部通过 JTAG 实现调试功能,并通过在开发板上引出的调试端口发送调试命令和接收调试信息,完成调试过程。JTAG 通过边界扫描技术实现对芯片输入输出信号的观察和控制。

3. 软件的固化与下载

小批量调试生产可通过调试器(JTAG 或 SWD 连接)进行 flash 的烧写固化,或者通过更改 BOOT 配置,利用 MCU 中内嵌的 bootloader 程序进行串行 flash 烧写,当需要大批量生产时,可采用专用的 flash 烧写器。

由于 ARM 芯片的广泛使用,众多开发工具都支持 ARM 平台程序的开发,ARM 开发的编译器主要有 ARMCC 和 ARM-LINUX-GCC 两种,前者是 ARM 公司出品的编译器,完全符合 ARM 指令集格式,后者是基于 Linux 的 ARM 交叉编译器,使用的是 GNU 的汇编方式,除了指令集与 ARM 指令兼容外,还支持一些非 ARM 标准的语法。由于开发软件越来越复杂,因此集成开发环境整合了编辑器、汇编器、编译器、调试器、模拟器等功能,使得应

用系统的开发更为便捷。

ARM的集成开发环境主要有ADS,KEIL MDK和IAR EWARM等。ADS是ARM公司早期的集成开发环境,采用CodeWarrior IDE,并集成了ARM开发包和应用库。KEIL原本是单片机开发环境,被ARM收购后作为ADS的升级版本MDK推出,被广泛使用。第三方开发的集成开发环境中,IAR EWARM的编译效率和优化表现最为突出。

4.3.2　程序开发库CMSIS

在程序设计中,我们会大量使用到库函数,为了便于ARM Cortex系列处理器的程序设计,ARM定义了ARM Cortex微控制器软件接口标准CMSIS。CMSIS为开发者访问底层硬件提供了一个API接口,通过使用固件函数库,无需深入掌握底层硬件细节就可以对外设进行控制。CMSIS由ARM和芯片厂家提供,ARM提供独立于芯片的内核设备访问层、中间设备访问的通用方法以及外设访问接口,芯片厂家在CMSIS基础上提供针对自己芯片的外设接口库,包含了GPIO、TIMER、CAN、I2C、SPI、UART和ADC等所有标准外设,便于进行二次开发和应用,可以大大减少用户的程序编写时间,进而降低开发成本,缩短在不同处理器之间的移植时间。

CMSIS为Cortex-M微控制器系统定义了:

- 访问外设寄存器的通用方法和定义异常向量的通用方法。
- 内核设备的寄存器名称和内核异常向量的名称。
- 独立于微控制器的RTOS接口,带调试通道。
- 中间设备组件接口(TCP/IP协议栈,闪存文件系统)。

如图4-14所示,CMSIS由4部分组成:

- CMSIS-CORE组件:提供Cortex-M系列微处理器的内核和外设的寄存器定义、中断接口和API函数,提供系统启动方法和访问特定处理器功能和内核外设的函数。

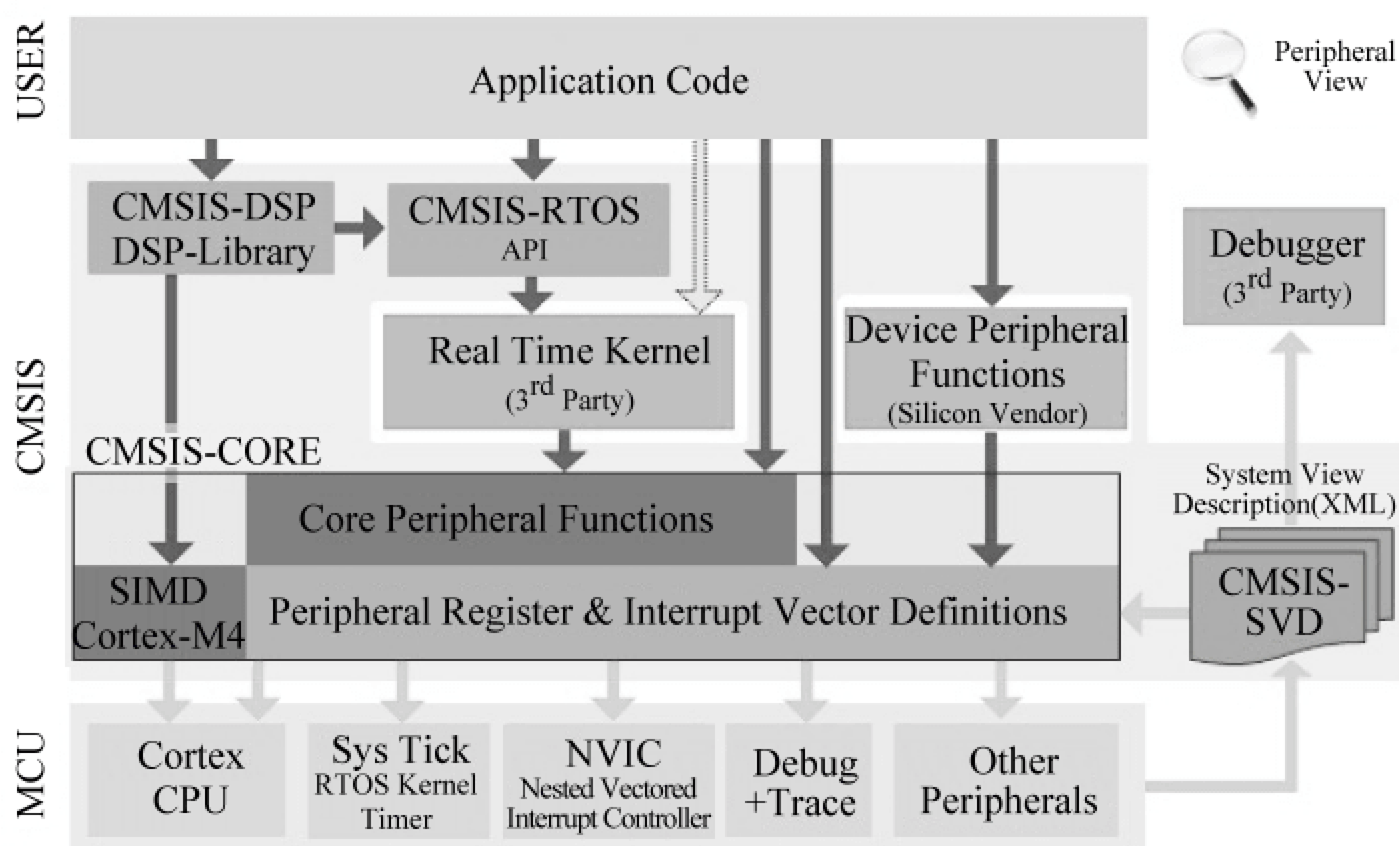


图4-14　CMSIS库的作用

• CMSIS-DSP 组件：优化的信号处理算法。
• CMSIS-SVD 组件：描述设备外设和中断的 XML 文件。
• CMSIS-RTOS API 组件：提供通用的 API 接口给一些实时操作系统(RTOS)。

CMSIS 提供的独立于编译器的主要内核文件包括：

• Cortex-M3 内核及其设备文件(core_cm3. h ＋ core_cm3. c)：用于访问 Cortex-M3 内核及其 NVIC,SysTick 等设备,访问 Cortex-M3 CPU 寄存器和内核外设的函数。
• 微控制器专用头文件(device. h)：用于指定中断号码(与启动文件一致),定义外设寄存器（寄存器的基地址和布局)。
• 微控制器专用系统文件(system_device. c)：包括 SystemInit(),SystemFrequncy(),Sysem_ExtMemCtl()等函数,用来初始化处理器。

CMSIS 提供的与编译器相关的启动文件主要是编译器启动代码（startup_device. s),它定义了中断处理程序列表及中断处理程序默认函数。

基于 CMSIS,ST 提供了 STM32Lx 系列的内核设备访问 API 和外围设备访问 API,可在 http：//www. st. com/web/en/catalog/tools/PF257908 下载,该库提供了 STM32L 系列中高、中、低密度处理器的编译器相关启动代码,内核设备文件、MCU 系统文件,以及外围控制器的所有驱动程序。

STM32Lx 系列的 CMSIS 库的结构如图 4-15 所示,CMSIS/Device/ST/STM32Lxx 目

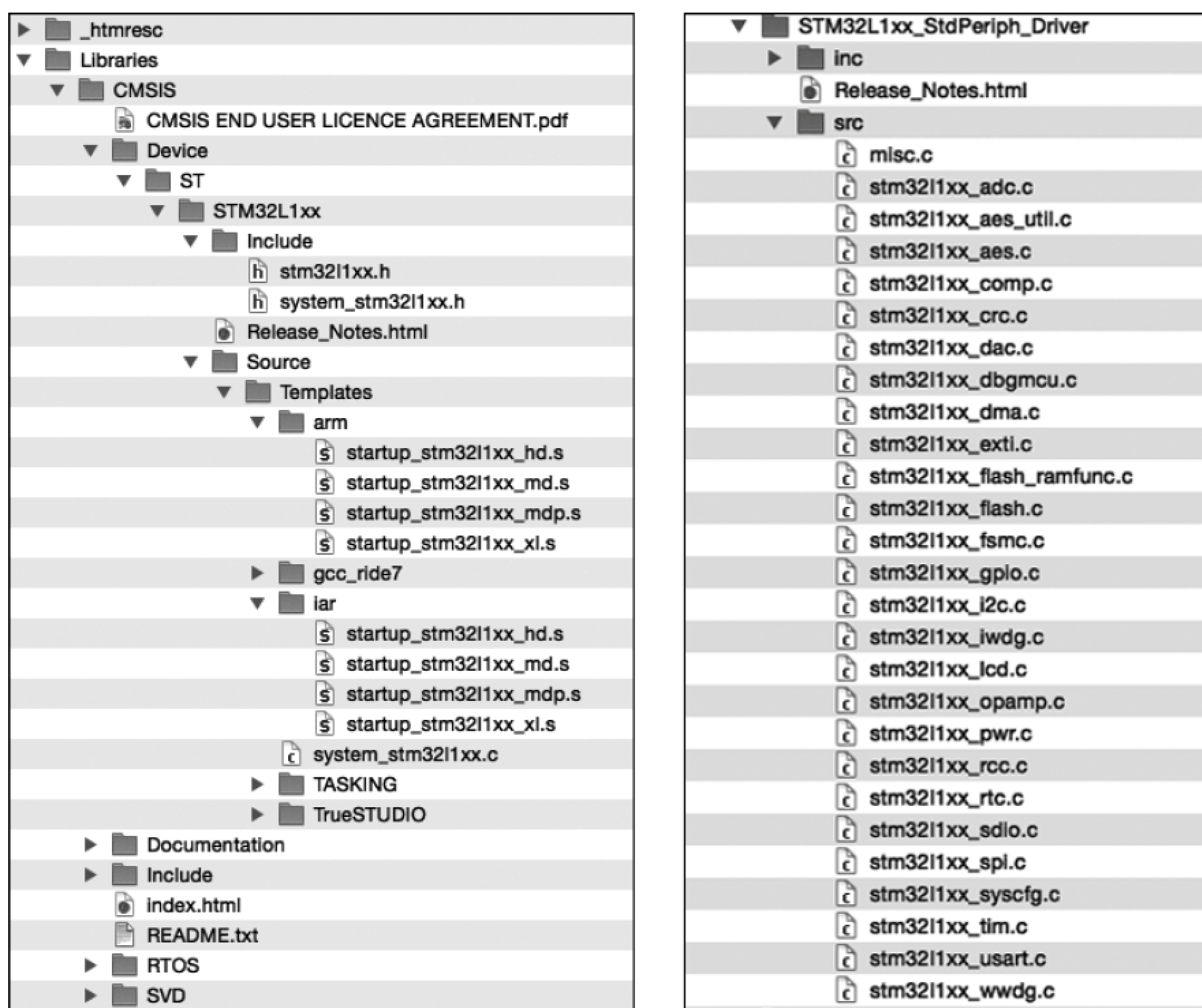


图 4-15　CMSIS 代码树

录下的 include 文件夹包含微控制器专用头文件 stm32l1xx. h 和微控制器专用系统文件的头文件 system_stm32l1xx. h;source/Templates 文件夹下包含的是微控制器专用系统文件 system_ stm32l1xx. c 和编译器相关启动文件,其中 ARM 目录下是 ARMCC 编译器(KEILMDK 开发环境)的启动文件;IAR 目录下是 IAR EWARM 开发环境的启动文件,根据微控制器 flash 容量大小的不同,选用不同的启动文件。STM32L152-Discovery 开发板使用的 STM321L152RBT6 属于中密度型,因此在构建工程时选择 startup_stm32l1xx_md. s。

STM32L1xx_StdPeriph_Driver /src 目录下为 STM32L1xx 系列微控制器的外设驱动程序,包括 ADC、AES、Flash、LCD、GPIO 等。用户利用 ST 提供的 CMSIS 库可以方便地调用这些驱动程序实现快速开发。

此外,为便于开发,ST 还提供了 STM32LCube 库,实现了网络协议栈、USB 设备操作、图形化界面、文件系统、实时操作系统等接口。

4.3.3　STM32L52 嵌入式程序开发预备知识

1. C 语言的位操作

位操作是嵌入式系统中对于外围控制器寄存器操作的重要方法,本节对 C 语言中的位操作进行概要介绍。

C 语言支持 6 种位操作:

(1) &:按位"与",对两个操作数的每一位进行逻辑与操作,例如:

```
1000 1000 & 1000 0001 = 1000 0000;
```

(2) |:按位"或",对两个操作数中的每一位进行逻辑或操作,例如:

```
1000 1000 | 1000 0001 = 1000 1001;
```

(3) ^:按位"异或",对两个操作数中的每一位进行逻辑异或操作,仅当两个操作数不同时,相应的输出结果才为 1,否则为 0,例如:

```
1000 1000 ^ 1000 0001 = 0000 1001;
```

(4) ~:按位"取反",将操作数中地每一位取反,例如:

```
~1000 1000 = 0111 0111;
```

(5) <<:"算术左移"操作,将操作数的各位按要求向左移动若干位,例如 5<<3 等价于 00000101<<3,即 0010 1000。算术左移相当于乘法,左移 n 位的结果等于原操作数乘 2 的 n 次方。

(6) >>:"算术右移",将操作数的各位按要求向右移动若干位,例如 4>>2 等价于 0000 0100>>2,即 0000 0001。算术右移相当于除法,右移 n 位的结果等于原操作数除 2 的 n 次方。

位运算符主要用于快速乘除运算和寄存器操作。

(1) 快速乘除运算：移位操作可用于整数的快速乘除运算，左移一位等效于乘 2，而右移一位等效于除以 2。

如：x ＝ 7，二进制表达为：0000 0111，

```
x<<1          0000 1110,相当于：x=2*7=14,
x<<3          0111 0000,相当于：x=14*2*2*2=112
x<<2          1100 0000,相当于：x=112*2*2=448-256=192
```

在作第三次左移时，其中一位为 1 的位移到外面去了，而左边只能以 0 补齐，因而便不等于 112＊2＊2＝448，而是等于 192 了。当 x 按刚才的步骤反向移动回去时，就不能返回到原来的值了，因为左边丢掉的一个 1，再也不能找回来了：

```
x>>2          0011 0000,相当于 x=192/4=48
x>>3          0000 0110,相当于 x=48/8=6
x>>1          0000 0011   相当于 x=6/2=3
```

(2) 寄存器位操作：寄存器置指定位置为 1：PORTA |＝ (1<<n)，PORTA 的第 n 为置为 1，其他位不变。例如：

```
PORTA |= (1<<4) :将第四位置 1:
PORTA |= (1<<7) | (1<<4) | (1<<0) 将设第 7、4 和 0 位置 1
```

将寄存器指定位置为 0：PORTA &＝ ～(1<<n)，寄存器的第 n 位将被清 0，但不影响其他位，例如：

```
PORTA &=~(1<<4):第四位置 0
```

2. 用 C 语言操作硬件

第 3 章 3.3 节的启动代码分析中，我们已经可以实现从 MCU 上电复位到跳转到 C 语言的 main 函数开始执行，这样我们可以使用 C 语言对硬件进行控制。对于 C 程序，程序编译时已经指定了 Flash 的地址和 SRAM 的地址，因此建立堆栈后，数据处理类的程序即可正常运行。但通常我们要操作外设，配置外设的寄存器控制外设运行，从第 2 章的存储空间映射可知外设空间和 Flash、SRAM 统一遍址，对外设寄存器的访问就是对存储空间的访问。

Cortex-M3 的 SysTick 定时器的当前值寄存器地址为 0xE000E018，用 C 语言访问这个地址的一个字，即可读写该寄存器的值。

```
unsigned int *p=(unsigned int *)(0xE000E018)
unsigned int SysTick_Value=*p;          //读取 SysTick 计数器值
*p=SysTick_Value+2000;                  //向 SysTick 计数器写入新值
```

利用宏定义，可以给每个寄存器起一个名字。

```
#define SYSTICK_VALUE_R   (*(unsigned int*)0xE000E018)
```

这样可以直接使用寄存器名：

```
SYSTICK_VALUE_R = 20000;
```

通常我们将寄存器定义为如下形式：

```
#define SYSTICK_VALUE_R  (*(volatile unsigned int*)0xE000E018)
```

volatile是C的一个关键字，别称易失变量，即容易丢失的变量；因为编译器为了程序的效率，在编译时会进行一些优化。在变量前加上个volatile关键字，编译器就不会对该变量进行优化了，这样可以保证读取的是存储器地址而不是缓存或寄存器（优化后可能把该变量的值存放在某个临时的寄存器中，这样会导致寄存器和存储器内容不一致）。

再结合C语言的位操作运算，就可以对寄存器的任意bit进行访问和控制。CMSIS提供了寄存器的规范定义和HAL硬件抽象层的C库版本。

3. 时钟树及时钟配置

STM32L152的时钟树如图2-17所示。主要时钟源有以下5个：

1）HSE时钟

高速外部时钟信号(HSE)可以由HSE外部晶体/陶瓷谐振器和HSE用户外部时钟产生。为了减少时钟输出的失真和缩短启动稳定时间，晶体/陶瓷谐振器和负载电容器要尽可能地靠近振荡器引脚。负载电容值必须根据所选择的振荡器来调整。外部晶体/陶瓷谐振器可以提供一个非常精确的时钟源，HSE晶体可以通过设置时钟控制寄存器里RCC_CR中的HSEON位被启动和关闭。

2）HSI时钟

HSI时钟信号由内部16MHz的RC振荡器产生，可直接作为系统时钟或作为PLL输入。HSI RC振荡器能够在不需要任何外部器件的条件下提供系统时钟。它的启动时间比HSE晶体振荡器短。然而，即使在校准之后它的时钟频率精度仍较差。HSI RC可由时钟控制寄存器中的HSION位来启动和关闭。

3）LSE时钟

LSE晶体是一个32.768kHz的低速外部晶体或陶瓷谐振器。它为实时时钟或者其他定时功能提供一个低功耗的精确时钟源。LSE晶体通过在备份域控制寄存器(RCC_BDCR)里的LSEON位启动和关闭。

4）LSI时钟

LSI RC担当一个低功耗时钟源的角色，它可以在停机和待机模式下保持运行，为独立看门狗和自动唤醒单元提供时钟。LSI时钟频率为37kHz。LSI RC可以通过控制/状态寄存器(RCC_CSR)里的LSION位来启动或关闭。

5）MSI时钟

MSI是内部集成的多速率时钟，有65.5kHz、131kHz、262kHz、524kHz、1.05MHz、2.1MHz和4.2MHz 7种配置。

输入时钟源的频率一般都比较低，系统所需的时钟频率可能会比较高，因此我们可以采用PLL锁相环对输入时钟进行倍频处理。内部PLL可以用来倍频HSI的RC振荡时钟或

HSE 晶体时钟。PLL 的设置必须在其被激活前完成。一旦 PLL 被激活，这些参数就不能被改动。在使用 USB 时，PLL 必须被设置为输出 48 MHz 时钟提供 USBCLK 时钟。

对于微控制器系统，所需的时钟包括系统主时钟 SYSCLK，用于 Cortex-M3 处理器核心；AHB 总线时钟 HCKL，用于 AHB 总线和连接到 AHB 的外设；APB1 总线时钟 PPB1 和 APB2 总线时钟 PPB2，用于外围总线和连接到外围总线的外设。此外 USB、RTC 需要特殊的时钟需要配置，微控制器还提供一个输出 MCO，可以将 MCU 内部的时钟输出到芯片引脚上。

系统复位后，2.1MHz 的 MSI 被选为系统时钟。当时钟源被直接或通过 PLL 间接作为系统时钟时，它将不能被停止。只有当目标时钟源准备就绪时才可以进行系统时钟切换。时钟控制寄存器(RCC_CR)里的状态位指示哪个时钟已经准备好了，哪个时钟目前被用作系统时钟。

4. 时钟和复位配置控制器

1）时钟控制寄存器 RCC_CR

时钟控制寄存器用于配置内部和外部的高速时钟以及锁相环 PLL 的开启控制，其有效域定义如图 4-16 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	RTCPRE[1:0]		CSS ON	Reserved		PLL RDY	PLLON	Reserved					HSE BYP	HSE RDY	HSE ON
	rw	rw	rw			r	rw						rw	r	rw
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Reserved						MSI RDY	MSION	Reserved						HSI RDY	HSION
						r	rw							r	rw

图 4-16　时钟控制寄存器

HSION、MSION、HSEON、PLLON 分别用于控制 HSI、MSI、HSE 和 PLL 是否启用，若启用其中一个时钟源，其稳定后，对应的 HSIRDY、HSERDY、PLLDRY 和 MSIRDY 会自动置 1，表明该时钟可用。

该寄存器的初值为 0b0XX0 0000 0000 0X00 0000 0011 0000 0000，即默认 MSION 开启，MCU 上电后采用 MSI 时钟。

2）内部时钟源及时钟校准寄存器 RCC_ICSCR

内部时钟源及时钟校准寄存器的有效域定义如图 4-17 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MSITRIM[7:0]								MSICAL[7:0]							
rw	rw	rw	rw	rw	rw	rw	rw	r	r	r	r	r	r	r	r
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
MSIRANGE[2:0]			HSITRIM[4:0]					HSICAL[7:0]							
rw	rw	rw	rw	rw	rw	rw	rw	r	r	r	r	r	r	r	r

图 4-17　内部时钟源及时钟校准寄存器

MCU 上电后采用 MSI 时钟，MSI 为多速率时钟，RCC_ICSCR 寄存器的 MSIRANGE 域用于指定 MSI 时钟的频率，000 表示 65.536kHz，001 表示 131.072kHz，010 表示

262.144kHz,011 表示 524.288kHz,100 表示 1.048MHz,101 表示 2.097MHz(默认值),110 表示 4.194MHz,111 无效。该寄存器默认值为 0x00XX B0XX,即 MSIRANGE 配置为 101,MSI 频率 2.097MHz。

3) 时钟配置寄存器 RCC_CFGR

时钟配置寄存器用于配置 PLL,AHB、APB 的总线时钟,其有效域定义如图 4-18 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	MCOPRE[2:0]			Res.	MCOSEL[2:0]			PLLDIV[1:0]		PLLMUL[3:0]				Res.	PLL SRC
	rw	rw	rw		rw	rw	rw	rw	rw	rw	rw	rw	rw		rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved		PPRE2[2:0]			PPRE1[2:0]			HPRE[3:0]				SWS[1:0]		SW[1:0]	
		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	r	r	rw	rw

图 4-18　时钟配置寄存器

MCOPRE[2:0],MCU 输出时钟 MCO 的分频因子,000～100 分别表示 1、2、4、8、16 分频,其余配置无效。

MCOSEL[2:0]用于配置 MCO 输出时钟的时钟源,000 表示 MCO 输出禁用。001 表示 SYSCLK 作为 MCO 的输出源,010 表示 HSI 作为 MCO 的输出源,011 表示 MSI 作为 MCO 的输出源,100 表示 HSE 作为 MCO 的输出源,101 表示 PLL 作为 MCO 的输出源,110 表示 LSI 作为 MCO 的输出源,111 表示 LSE 作为 MCO 的输出源。

PLLDIV[1:0],PLL 时钟的分频系数,00 表示不分频,01～11 分别表示 2、3、4 分频。

PLLMUL[3:0]:PLL 的倍频系数,0000～1000 分别表示 3、4、6、8、12、16、24、32、48 倍频,其余配置无效。

PLLSRC,PLL 的输入时钟选择,0 表示 HIS,1 表示 HSE。

PPRE2[2:0]和 PPRE1[2:0]分别用于配置 APB2、APB1 的总线时钟分频系数,APB 的时钟来源于 AHB 的 HCLK,0xx 表示 HCLK 不分频,100～111 分别表示 2、4、8、16 分频。

HPRE[3:0],AHB 时钟分频系数,AHB 的时钟来源于 SYSCLK,0xxx 表示 SYSCLK 不分频,1000～1111 分别表示 2、4、8、16、64、128、256 和 512 分频。

SW[1:0],SYSCLK 来源配置,00 表示 MSI 作为系统时钟,01 示 HSI 作为系统时钟,10 表示 HSE 作为系统时钟,11 表示 PLL 作为系统时钟。

SWS[1:0],系统时钟的状态,只读,用于表示那个时钟源正在被作为系统时钟,00～11 分别表示 MSI、HSI、HSE 和 PLL。

4) AHB 外围时钟使能寄存器 RCC_AHBENR

AHB 外围时钟使能寄存器用于配置连接到 AHB 总线上的每个外设时钟是否启用,其有效域定义如图 4-19 所示,写 1 表示启用,写 0 表示关闭时钟。

5) APB2 外围时钟使能寄存器 RCC_APB2ENR

APB2 为时钟使能寄存器,用于配置连接到 APB2 总线上的每个外设时钟是否启用,其有效域定义如图 4-20 所示,写 1 表示启用,写 0 表示关闭时钟。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	FSMC EN	Reserved		AES EN	Res.	DMA2EN	DMA1EN	Reserved							
	rw			rw		rw	rw								
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
FLITF EN	Reserved		CRCEN	Reserved				GPIOG EN	GPIOF EN	GPIOH EN	GPIOE EN	GPIOD EN	GPIOC EN	GPIOB EN	GPIOA EN
rw			rw					rw	rw	rw	rw	rw	rw	rw	rw

图 4-19　AHB 外围时钟使能寄存器

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Res.	USART1 EN	Res.	SPI1 EN	SDIO EN	Res.	ADC1 EN	Reserved				TIM11 EN	TIM10 EN	TIM9 EN	Res.	SYSCF GEN
	rw		rw	rw		rw					rw	rw	rw		rw

图 4-20　APB2 外围时钟使能寄存器

6）APB1 外围时钟使能寄存器 RCC_APB1ENR

APB1 时钟使能寄存器用于配置连接到 APB1 总线上的每个外设时钟是否启用，其有效域定义如图 4-21 所示，写 1 表示启用，写 0 表示关闭时钟。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
COMP EN	Res.	DAC EN	PWR EN	Reserved				USB EN	I2C2 EN	I2C1 EN	UART5 EN	UART4 EN	USART3 EN	USART2 EN	Res.
rw		rw	rw					rw	rw	rw	rw	rw	rw	rw	
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
SPI3 EN	SPI2 EN	Reserved		WWD GEN	Res.	LCD EN	Reserved			TIM7 EN	TIM6 EN	TIM5 EN	TIM4 EN	TIM3 EN	TIM2 EN
rw	rw			rw		rw				rw	rw	rw	rw	rw	rw

图 4-21　APB1 外围时钟使能寄存器

7）控制/状态寄存器 RCC_CSR

控制和状态寄存器的有效域定义如图 4-22 所示。RCC_CSR 中涉及 LSI 和 LSE 的启用和配置，分别用 LSION、LSEON、LSIRDY、LSERDY 表示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
LPWR RSTF	WWDG RSTF	IWDG RSTF	SFT RSTF	POR RSTF	PIN RSTF	OBLRS TF	RMVF	RTC RST	RTC EN	Reserved				RTCSEL[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw					rw	rw
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Reserved			LSECS SD	LSECS SON	LSE BYP	LSERDY	LSEON	Reserved						LSI RDY	LSION
			r	rw	rw	r	rw							r	rw

图 4-22　控制/状态寄存器

8）时钟配置的相关寄存器和库函数

（1）寄存器结构定义

CMSIS 中 RCC 寄存器结构定义为 RCC_TypeDeff，在文件“stm32L1xx.h”中定义

如下：

```
typedef struct
{
    __IO uint32_t CR;                                //时钟控制寄存器
    __IO uint32_t CFGR;                              //时钟配置寄存器
    __IO uint32_t CIR;                               //时钟中断寄存器
    __IO uint32_t AHBRSTR                            //AHB 外设复位寄存器
    __IO uint32_t APB2RSTR;                          //APB2 外设复位寄存器
    __IO uint32_t APB1RSTR;                          //APB1 外设复位寄存器
    __IO uint32_t AHBENR;                            //AHB 外设时钟使能寄存器
    __IO uint32_t APB2ENR;                           //APB2 外设时钟使能寄存器
    __IO uint32_t APB1ENR;                           //APB1 外设时钟使能寄存器
    __IO uint32_t AHBLPENR                           //AHB 低功耗模式使能寄存器
    __IO uint32_t  APB2LPENR                         //APB2 低功耗模式使能寄存器
    __IO uint32_t APB1LPENR                          //APB1 低功耗模式使能寄存器
    __IO uint32_t CSR;                               //控制/状态寄存器
} RCC_TypeDef;
```

RCC 外设的寄存器地址定义在文件“stmL1xx.h”：

```
#define PERIPH_BASE ((uint32_t)0x40000000)
#define APB1PERIPH_BASE PERIPH_BASE
#define APB2PERIPH_BASE (PERIPH_BASE + 0x10000)
#define AHBPERIPH_BASE (PERIPH_BASE + 0x20000)
#define RCC_BASE (AHBPERIPH_BASE + 0x1000)
#define RCC ((RCC_TypeDef * ) RCC_BASE)
```

（2）配置案例

```
static void SetSysClock(void)
{
    __IO uint32_t StartUpCounter = 0, HSEStatus = 0;
    RCC->CR |= ((uint32_t)RCC_CR_HSEON);                    //使能 HSE
    do                                                       //等待 HSE 工作或超时
    {
        HSEStatus = RCC->CR & RCC_CR_HSERDY;
        StartUpCounter++;
    } while((HSEStatus == 0) && (StartUpCounter != HSE_STARTUP_TIMEOUT));
    if ((RCC->CR & RCC_CR_HSERDY) != RESET)
        HSEStatus = (uint32_t)0x01;
    else
        HSEStatus = (uint32_t)0x00;
    if (HSEStatus == (uint32_t)0x01)                         //HSE 正常工作
    {
```

```
        //配置 CFG 寄存器,HCLK、APB1、APB2 分频系数均为 1
        RCC->CFGR |= (uint32_t)RCC_CFGR_HPRE_DIV1;              //HCLK = SYSCLK/1
        RCC->CFGR |= (uint32_t)RCC_CFGR_PPRE2_DIV1;             //PCLK2 = HCLK/1
        RCC->CFGR |= (uint32_t)RCC_CFGR_PPRE1_DIV1;             //PCLK1 = HCLK/1
        //清除 PLL 控制位,配置 PLL 的时钟源、倍频系数、分频系数
        RCC->CFGR &= (uint32_t)((uint32_t)~(RCC_CFGR_PLLSRC | RCC_CFGR_PLLMUL |
                                        RCC_CFGR_PLLDIV));
        RCC->CFGR |= (uint32_t)(RCC_CFGR_PLLSRC_HSE | RCC_CFGR_PLLMUL12 |
                                RCC_CFGR_PLLDIV3);
        RCC->CR |= RCC_CR_PLLON;                                //启用 PLL
        while((RCC->CR & RCC_CR_PLLRDY) == 0) ;                 //等待 PLL 稳定
        //配置 CFGR 的 SW 选择 PLL 时钟作为系统时钟源
        RCC->CFGR &= (uint32_t)((uint32_t)~(RCC_CFGR_SW));
        RCC->CFGR |= (uint32_t)RCC_CFGR_SW_PLL;
        //读 CFGR 的 SWS,等待系统时钟状态指示 PLL 时钟成为系统时钟
        while ((RCC->CFGR&(uint32_t)RCC_CFGR_SWS) != (uint32_t)RCC_CFGR_SWS_PLL) ;
    }
    else
        //HSE 不工作,使用原有时钟即可
}
```

(3) RCC 库函数

RCC 库函数如表 4-1 所示。

表 4-1　为 ST 提供的 RCC 控制相关库函数

函　数　名	功　　能
RCC_DeInit	将外设 RCC 寄存器重设为默认值
RCC_HSEConfig	设置外部高速晶振(HSE)
RCC_WaitForHSEStartUp	等待 HSE 起振
RCC_HSICmd	使能或者失能内部高速晶振(HSI)
RCC_PLLConfig	设置 PLL 时钟源及倍频系数
RCC_PLLCmd	使能或者失能 PLL
RCC_SYSCLKConfig	设置系统时钟(SYSCLK)
RCC_GetSYSCLKSource	返回用作系统时钟的时钟源
RCC_HCLKConfig	设置 AHB 时钟(HCLK)
RCC_PCLK1Config	设置低速 AHB 时钟(PCLK1)
RCC_PCLK2Config	设置高速 AHB 时钟(PCLK2)
RCC_ITConfig	使能或者失能指定的 RCC 中断

续表

函　数　名	功　　能
RCC_ADCCLKConfig	设置 ADC 时钟（ADCCLK）
RCC_LSEConfig	设置外部低速晶振（LSE）
RCC_LSICmd	使能或者失能内部低速晶振（LSI）
RCC_RTCCLKConfig	设置 RTC 时钟（RTCCLK）
RCC_RTCCLKCmd	使能或者失能 RTC 时钟
RCC_GetClocksFreq	返回不同片上时钟的频率
RCC_AHBPeriphClockCmd	使能或者失能 AHB 外设时钟
RCC_APB2PeriphClockCmd	使能或者失能 APB2 外设时钟
RCC_APB1PeriphClockCmd	使能或者失能 APB1 外设时钟
RCC_GetFlagStatus	检查指定的 RCC 标志位设置与否
RCC_GetITStatus	检查指定的 RCC 中断发生与否
RCC_ClearITPendingBit	清除 RCC 的中断待处理位

配置示例：

```
void RCC_Configuration(void)
{
  RCC_DeInit();
  RCC_HSEConfig(RCC_HSE_ON);                              //RCC_HSE_ON——HSE 晶振打开(ON)
  HSEStartUpStatus = RCC_WaitForHSEStartUp();
  if(HSEStartUpStatus == SUCCESS)                         //SUCCESS:HSE 晶振稳定且就绪
  {
    RCC_HCLKConfig(RCC_SYSCLK_Div1)                       //AHB 时钟为系统时钟
    RCC_PCLK2Config(RCC_HCLK_Div1);                       //APB2 时钟为 HCLK
    RCC_PCLK1Config(RCC_HCLK_Div2);                       //APB1 时钟为 HCLK / 2
    RCC_PLLConfig(RCC_PLLSource_HSE_Div1, RCC_PLLMul_4);
    //PLL 的输入时钟 = HSE 时钟频率;RCC_PLLMul_4——PLL 输入时钟 x 4
    RCC_PLLCmd(ENABLE);
    while(RCC_GetFlagStatus(RCC_FLAG_PLLRDY) == RESET) ;
    RCC_SYSCLKConfig(RCC_SYSCLKSource_PLLCLK);
    //RCC_SYSCLKSource_PLLCLK——选择 PLL 作为系统时钟
    while(RCC_GetSYSCLKSource() != 0x08);                 //0x08:PLL 作为系统时钟
  }
  //启动 GPIOA、GPIOB 和 GPIOC 外设时钟
  RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA | RCC_AHBPeriph_GPIOB |
                        RCC_AHBPeriph_GPIOC, ENABLE);
}
```

Chapter 5

第5章　通用输入输出

【导读】 通用输入输出(General Purpose Input/Output, GPIO)是嵌入式微控制器最常用、最基础、灵活性最强的控制器,可以实现简单控制,也可以组合实现复杂时序,是嵌入式程序设计必须掌握的控制器。本章首先介绍 GPIO 的引脚内部构造,不同的输入输出模式的区别,然后对 GPIO 的寄存器定义进行了详细介绍,并结合 ST 的外围控制器库函数对典型 API 进行了介绍,最后以 LED 和按键为例阐述了利用库函数进行 I/O 控制的方法。

5.1　GPIO 原理

5.1.1　GPIO 功能

GPIO 是嵌入式开发里面最基本也最常用的硬件端口,STM32L1xx 系列处理器可提供多达 128 个 I/O,实现输入输出功能,并将其分为 A～H 8 组,每组称为一个端口(PORT),每个端口有 16 个 I/O 引脚(PIN),每个 I/O 的速度可单独配置,最快可在 2 个时钟周期进行 I/O 翻转,支持 I/O 锁定功能、I/O 复选功能,每个引脚最多可有 16 个复选功能,最大程度的提供了灵活的 I/O 配置和使用。

I/O 的使用由 GPIO 控制器管理,每个 GPIO 端口有四个 32 位配置寄存器(GPIOx_MODER, GPIOx_OTYPER, GPIOx_OSPEEDR 和 GPIOx_PUPDR),两个 32 位数据寄存器(GPIOx_IDR 和 GPIOx_ODR),一个 32 位置位/复位寄存器(GPIOx_BSRR),一个 32 位锁定寄存器(GPIOx_LCKR)和两个 32 位的复选功能选择寄存器(GPIOx_AFRH 和 GPIOx_AFRL)。

一个 I/O 端口引脚的结构如图 5-1 所示,主要由输入驱动器、输出驱动器、输入数据寄存器、输出数据寄存器和位设置/清除寄存器进行输入输出控制。

GPIO 端口的每个位可以由软件分别配置成多种模式: 输入、输出、复用和模拟四种模式,每种模式下,可以通过寄存器对输入输出方式进行配置,由输入驱动器和输出驱动器的开关电路控制,每个 I/O 端口位可以自由编程,以 32 位字访问 I/O 端口寄存器。

在每个 AHB 总线时钟周期,GPIO 控制器采样 I/O 引脚的输入电平,并将输入数据存储到输入寄存器中。所有 GPIO 引脚有一个内部弱上拉和弱下拉,当配置为输入时,它们可以被激活也可以被断开;端口配置为输出模式时,GPIO 输出寄存器中的值输出到对应的

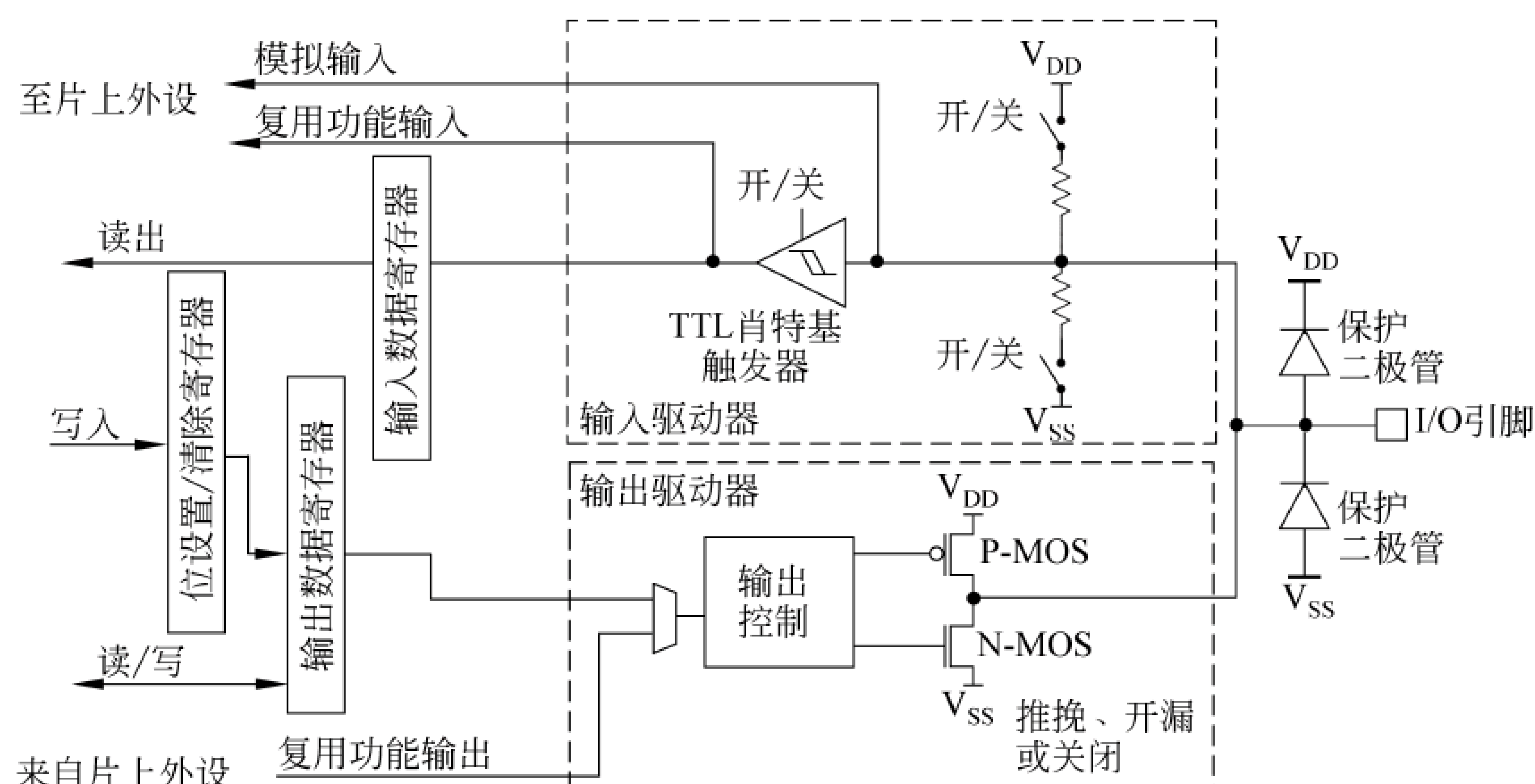


图 5-1　I/O 引脚内部电路结构

I/O 引脚上。

为便于程序对 I/O 口的输出数据寄存器控制，GPIO 控制器提供了单个或多个位的原子读写操作，通过对“置位/复位寄存器”(GPIOx_BSRR)中想要更改的位写 1 实现同时操作，无需软件进行开关中断的保护操作。

为保护 I/O 引脚配置的安全性，GPIO 控制器提供了锁定机制允许冻结 I/O 配置。当在一个端口位上执行了锁定(LOCK)程序，在下一次复位之前，将不能再更改端口位的配置。

所有的 I/O 端口及其引脚都具有复用功能，复用功能是将 GPIO 端口映射到某个外围控制器上，作为外围控制器的专用 I/O 通道，这样，可以对不使用的外围控制器的引脚当作普通 I/O 使用，当需要时配置成为专用 I/O 使用。当外围控制器功能不能满足时，我们通常使用 I/O 模拟专用控制器的时序用软件实现专用控制器的功能。每个 I/O 端口最多可以有 16 个复选功能 AF0-AF15，但只能使用一个复用功能。CPU 上电复位后，所有的 I/O 都默认使用 AF0 功能。除了特殊 I/O 引脚外(比如 JTAG 调试口(PA15、PA14、PA13、PB4、PB3)对应的 I/O 引脚 AF0 功能为 JTAG 调试端口功能，因此每个引脚被置为输入上拉或下拉模式)，I/O 引脚的 AF0 功能都是 GPIO，复位后的默认配置为浮空输入。为了使不同器件封装的外设 I/O 功能的数量达到最优，STM32L1xx 系列处理器支持复用功能重映射，可以把一些复用功能的引脚重新映射到其他一些引脚上。

所有的 GPIO 端口都有外部中断能力，将端口配置为输入模式后，可以把引脚作为中断源的输入，例如按键、事件触发等，具体在中断控制器部分进行阐述。

5.1.2　I/O 模式配置

1. 输入配置

当 I/O 端口配置为输入时：图 5-1 中的输出缓冲器被禁止，施密特触发输入被激活。

根据输入引脚配置(上拉、下拉或浮空)的不同,弱上拉或下拉电阻被连接。出现在I/O引脚上的数据被采样到输入数据寄存器,对输入数据寄存器的读访问可得到I/O引脚的状态。

浮空输入:浮空输入一般多用于外部按键输入,浮空输入状态下,I/O的电平状态是不确定的,完全由外部输入决定。

上拉输入和下拉输入分别对应图5-1中的上拉电阻开关和下拉电阻开关,将输入信号的默认值保持在高电平或者低电平。

2. 输出配置

当I/O端口被配置为输出时,图5-1中的输出缓冲器被激活,施密特触发输入被激活,弱上拉或下拉电阻被禁止。输出到I/O引脚上的数据在每个时钟周期同时被采样到输入数据寄存器,在开漏模式时,对输入数据寄存器的读访问可得到输出I/O状态。在推挽式模式时,对输出数据寄存器的读访问得到当前的输出状态。

输出类型可配置为两种方式:开漏模式和推挽模式。开漏模式下,图5-1中的输出寄存器上的0激活N-MOS,而输出寄存器上的1将端口置于高阻状态(P-MOS从不被激活)。推挽模式下,图5-1中的输出寄存器上的0激活N-MOS,而输出寄存器上的1将激活P-MOS。

推挽输出:可以输出高、低电平,连接数字器件;推挽一般是指两个三极管分别受两个互补信号的控制,总是在一个三极管导通的时候另一个截止,形成推拉结构,所以导通损耗小、效率高。输出既可以向负载灌电流,也可以从负载抽取电流。推拉式输出级既提高电路的负载能力,又提高开关速度,高低电平由电源决定。

开漏输出:输出端相当于三极管的集电极,适合于做电流型的驱动,其吸收电流的能力相对强(20mA以内)。开漏电路利用外部电路的驱动能力,可以减少芯片内部的驱动电流,一般用来连接不同电平的器件。由于开漏引脚不连接外部的上拉电阻时,只能输出低电平,如果需要同时具备输出高电平的功能,则需要接上拉电阻,上升沿的时延由上拉电阻的阻值决定,电阻小时延时就小,功耗大;反之延时大功耗小。在多个I/O口形成“线与”功能时,需要设置为开漏输出。

3. 复用功能配置

当I/O端口被配置为复用功能时,图5-1中,在开漏或推挽输出配置中,输出缓冲器被打开,输出缓冲器由内置外设的信号驱动输出(复用功能输出);施密特触发输入被激活,弱上拉和下拉电阻由GPIOx_PUPDR寄存器确定,每个时钟周期采样I/O引脚上的数据,读输入数据寄存器时可得到I/O口状态。复用功能下I/O口的输入输出配置以及开漏、推挽由复用功能引脚规定。

对于复用的输入功能,端口必须配置成输入模式(浮空、上拉或下拉)且输入引脚必须由外部驱动。对于复用输出功能,端口必须配置成复用功能输出模式(推挽或开漏)。对于双向复用功能,端口位必须配置复用功能输出模式(推挽或开漏)。这时,输入驱动器被配置成浮空输入模式,引脚和输出寄存器断开,并和片上外设的输出信号连接。如果软件把一个GPIO脚配置成复用输出功能,但是外设没有被激活,它的输出将不确定。

4. 模拟配置

当I/O端口被配置为模拟时，图5-1中的输出缓冲器被禁止，施密特触发输入被禁止，I/O信号直接连接到处理器的模拟外围控制器电路，不经过GPIO控制器，施密特触发输出值被强置为0，弱上拉和下拉电阻被禁止，此时读取输入数据寄存器时数值为0。在低功耗休眠时，可将I/O设置为模拟输入状态，以降低功耗。

对于I/O引脚在不同应用情况下的配置如表5-1所示。

表5-1　I/O配置的典型配置

应用场景	配　　置
普通GPIO输入	配置该引脚为浮空输入、带弱上拉输入或带弱下拉输入，不使能该引脚对应的所有复用功能模块
普通GPIO输出	根据需要配置该引脚为推挽输出或开漏输出，不使能该引脚对应的所有复用功能模块
普通模拟输入	配置该引脚为模拟输入模式，不使能该引脚对应的所有复用功能模块
内置外设的输入	根据需要配置该引脚为浮空输入、带弱上拉输入或带弱下拉输入，使能该引脚对应的复用功能模块
内置外设的输出	根据需要配置该引脚为复用推挽输出或复用开漏输出，使能该引脚对应的复用功能模块。

5.2　GPIO寄存器

GPIO控制器的寄存器如表5-2所示。

表5-2　GPIO控制器寄存器

寄存器名称	偏移量	功　　能	复　位　值
端口模式寄存器(GPIOx_MODER)	0x00	配置端口的输入输出方向	端口A：0xA800 000 端口B：0x0000 0280 其他：0x0000 0000
端口输出类型寄存器(GPIOx_OTYPER)	0x04	配置端口输出类型	0x0000 0000
端口输出速度寄存器(GPIOx_OSPEEDR)	0x08	配置端口输出输速率	端口B：0x0000 00C0 其他：Ox0000 0000
端口上拉下拉寄存器(GPIOx_PUPDR)	0x0C	配置端口上拉和下拉电阻	端口A：0x6400 0000 端口B：0x0000 0100 其他：0x0000 0000
GPIO端口输入数据寄存器(GPIOx_IDR)	0x10	端口输入数据	0x0000 XXXX（X表示不定态）
GPIO端口输出寄存器(GPIOx_ODR)	0x14	端口输出数据	0x0000 0000

续表

寄存器名称	偏移量	功　　能	复　位　值
端口位设置/清除寄存器(GPIOx_BSRR)	0x18	端口输出寄存器设置为 1 或清为 0	0x0000 0000
GPIO 端口配置锁定寄存器(GPIOx_LCKR)	0x1C	锁定引脚的配置信息	0x0000 0000
复用功能低寄存器(GPIOx_AFRL)	0x20	复用功能选择 0～7	0x0000 0000
复用功能高寄存器(GPIOx_AFRH)	0x28	复用功能选择 8～15	0x0000 0000

1. GPIO 端口模式寄存器(GPIOx_MODER)(x＝A,…,H)

端口模式寄存器用于配置 I/O 的输入输出状态,其有效域定义如图 5-2 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 5-2　端口模式寄存器定义

端口模式寄存器是一个 32 位寄存器,每 2 个 bit 为一组,共定义了 16 个 I/O 的端口输入输出配置。MODERy[1：0]：端口 x 的 y 引脚配置位 (y=0,…,15),用于配置 I/O 口的输入输出方向。00 表示输入模式 (复位值),01 表示输出模式,10 表示复用功能模式,11 表示模拟模式。

2. GPIO 端口输出类型寄存器(GPIOx_OTYPER)(x＝A,…,H)

端口输出类型寄存器用于定义 I/O 引脚在输出模式下是开漏输出还是推挽输出,其有效域定义如图 5-3 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OT15	OT14	OT13	OT12	OT11	OT10	OT9	OT8	OT7	OT6	OT5	OT4	OT3	OT2	OT1	OT0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 5-3　端口输出类型寄存器定义

端口输出类型寄存器 32 位,其中高 16 位保留,低 16 位 OTy 表示端口 x 的引脚 y 的配置(y=0,…,15),0 表示推挽输出 (复位值),1 表示开漏输出。

3. GPIO 端口输出速度寄存器(GPIOx_OSPEEDR)(x＝A,…,H)

端口输出速度寄存器是一个 32 位寄存器,用于配置 I/O 引脚的输出速度(输出信号从低电平到高电平的上升速度或高电平到低电平的下降速度),每两个 bit 用于确定一个 I/O 引脚的输出速度配置,其有效域定义如图 5-4 所示。

OSPEEDRy[1：0]表示端口 x 配置 y 引脚配置(y=0,…,15),00 表示极低速 400kHz,

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
OSPEEDR15 [1:0]		OSPEEDR14 [1:0]		OSPEEDR13 [1:0]		OSPEEDR12 [1:0]		OSPEEDR11 [1:0]		OSPEEDR10 [1:0]		OSPEEDR9 [1:0]		OSPEEDR8 [1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OSPEEDR7[1:0]		OSPEEDR6[1:0]		OSPEEDR5[1:0]		OSPEEDR4[1:0]		OSPEEDR3[1:0]		OSPEEDR2[1:0]		OSPEEDR1 [1:0]		OSPEEDR0 1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 5-4　端口输出速度寄存器定义

01 表示低速 2MHz,10 表示中速 10MHz,11 表示高速 40MHz。

4. GPIO 端口上拉下拉寄存器(GPIOx_PUPDR)(x=A,…,H)

端口上拉下拉寄存器用于配置 I/O 引脚是否使用内部弱上拉和弱下拉电阻,其有效域定义如图 5-5 所示。每两位 PUPDRy[1:0]表示端口 x 的 y 引脚配置(y=0,…,15),00 表示浮空,01 表示上拉,10 表示下拉,11 保留。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PUPDR15[1:0]		PUPDR14[1:0]		PUPDR13[1:0]		PUPDR12[1:0]		PUPDR11[1:0]		PUPDR10[1:0]		PUPDR9[1:0]		PUPDR8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PUPDR7[1:0]		PUPDR6[1:0]		PUPDR5[1:0]		PUPDR4[1:0]		PUPDR3[1:0]		PUPDR2[1:0]		PUPDR1[1:0]		PUPDR0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 5-5　端口上拉下拉寄存器定义

5. GPIO 端口输入数据寄存器(GPIOx_IDR)(x = A,…,H)

输入数据寄存器用于存储端口输入引脚的电平值,其有效域定义如图 5-6 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

图 5-6　端口输入数据寄存器定义

端口输入数据寄存器 32 位中低 16 位有效,每位分别表示对应引脚的输入电平状态。IDRy 为端口 x 的引脚 y 的输入数据 (y=0,…,15),该寄存器为只读寄存器,以字为单位访问。

6. GPIO 端口输出寄存器(GPIOx_ODR)(x=A,…,H)

输出数据寄存器用于存储端口输出引脚的电平值,其有效域定义如图 5-7 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 5-7　端口输出数据寄存器定义

端口输出寄存器的高16位保留，低16位有效，每位表示对应引脚应该输出的电平状态。ODRy表示端口x的引脚y的输出数据(y=0,…,15)。该寄存器可读写，只能以字的形式操作。为便于操作，可通过配置GPIOx_BSRR寄存器对指定ODR位设置和清除。

7. GPIO端口位设置/清除寄存器(GPIOx_BSRR)(x = A,…,H)

端口位设置/清除寄存器用于修改数据输出寄存器的值，实现高效、原子操作的端口引脚输出电平配置，其有效域定义如图5-8所示。高16位对应16个I/O引脚置零操作，BRy表示清除端口x的引脚y(y = 0,…,15)，0表示对相应的ODRy位不产生影响，1表示清除相应的ODRy位为0。这些位只能以字、半字或字节写入。低16位对应16个同样的I/O引脚，BSy表示设置端口x的引脚y(y = 0,…,15)，0表示对相应的ODRy位不产生影响，1表示设置相应的ODRy位为1。读取该寄存器返回值为0。如果同时设置了BSy和BRy的对应位，BSy位起作用。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

图5-8　端口输出设置/清除寄存器定义

8. GPIO端口配置锁定寄存器(GPIOx_LCKR)(x = A,…,H)

配置锁定寄存器用于端口配置的锁定，其有效域定义如图5-9所示。锁定开关为寄存器的位16 LCKK，当LCKK为1时，根据该寄存器低16位LCKy的设置对对应I/O引脚进行锁定。当对相应的端口位执行了LOCK后，在下次系统复位之前将不能再更改端口位的配置。在执行锁定序列设置期间，不可以改变LCKP[15：0]的值，该寄存器只能以字操作。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															LCKK
															rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LCK15	LCK14	LCK13	LCK12	LCK11	LCK10	LCK9	LCK8	LCK7	LCK6	LCK5	LCK4	LCK3	LCK2	LCK1	LCK0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图5-9　端口锁定配置寄存器定义

LCKK：锁键(Lock key)，该位可随时读出，但只可通过锁键写入序列修改。该位置0表示端口配置锁键未激活，置1表示端口配置锁键被激活，下次系统复位前GPIOx_LCKR寄存器被锁住。

LCKK需要通过一个写入序列进行修改：写LCKK=1 -> 写LCKK=0 -> 写LCKK=1 -> 读LCKK -> 读LCKK=1。最后一个读可省略，但可以用来确认锁键已被激活。

LCKy 表示端口 x 的 y 引脚是否加锁(y = 0,…,15),0 表示不锁定端口的配置,1 表示锁定端口的配置。这些位可读可写但只能在 LCKK 位为 0 时写入。

被锁定的寄存器包括 GPIOx_MODER,GPIOx_OTYPER,GPIOx_OSPEEDR,GPIOx_PUPDR,GPIOx_AFRL 和 GPIOx_AFRH。

9. GPIO 复用功能低寄存器(GPIOx_AFRL)(x = A,…,H)

端口复用功能寄存器用于配置 I/O 端口作为其他功能使用,其有效域定义如图 5-10 所示和 5-11 所示。每个 4 个 bit 对应一个 I/O 引脚,每个引脚最多有 16 种复用功能。端口复用功能低寄存器用于配置一个 GPIO 端口的 0～7 引脚的复用功能。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFRL7[3:0]				AFRL6[3:0]				AFRL5[3:0]				AFRL4[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AFRL3[3:0]				AFRL2[3:0]				AFRL1[3:0]				AFRL0[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 5-10　端口复用功能低寄存器定义

AFRLy[3:0]:端口 x 的引脚 y 的复用功能启用位(y=0,…,7),AFRLy 配置的复用功能有 AFx 确定,取值如表 5-3 所示,每个引脚的 AF0～15 的具体配置见 STM32L152RE 的芯片数据手册(https://www.st.com/resource/en/datasheet/stm32l152re.pelf)

表 5-3　复用功能选择

AFRLy/AFRHy 取值	复 用 功 能							
AFRLy	0000	0001	0010	0011	0100	0101	0110	0111
	AF0	AF1	AF2	AF3	AF4	AF5	AF6	AF7
AFRHy	1000	1001	1010	1011	1100	1101	1110	1111
	AF8	AF9	AF10	AF11	AF12	AF13	AF14	AF15

10. GPIO 复用功能高寄存器(GPIOx_AFRH)(x = A,…,H)

复用功能高寄存器用于配置 I/O 口 8～15 的复用功能,其有效域定义如图 5-11 所示,AFRHy(y=8,…,15)的配置如表 5-3 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFRH15[3:0]				AFRH14[3:0]				AFRH13[3:0]				AFRH12[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AFRH11[3:0]				AFRH10[3:0]				AFRH9[3:0]				AFRH8[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 5-11　端口复用功能高寄存器定义

5.3 GPIO 操作函数库

CMSIS 中 GPIO 控制器的数据结构定义在 stm32lxx.h 中：

```
typedef struct
{
    __IO uint32_t MODER;        //模式寄存器,地址偏移量:0x00
    __IO uint16_t OTYPER;       //输出类型寄存器,地址偏移量:0x04
    uint16_t RESERVED0;         //保留,4字节对齐用
    __IO uint32_t OSPEEDR;      //输出速度寄存器,地址偏移量:0x08
    __IO uint32_t PUPDR;        //上拉下拉寄存器,地址偏移量:0x0C
    __IO uint16_t IDR;          //输入数据寄存器,地址偏移量:0x10
    uint16_t RESERVED1;         //保留,4字节对齐用
    __IO uint16_t ODR;          //输出数据寄存器,地址偏移量:0x14
    uint16_t RESERVED2;         //保留,4字节对齐用
    __IO uint16_t BSRRL;        //位设置/清除低寄存器,地址偏移量:0x18
    __IO uint16_t BSRRH;        //位设置/清除高寄存器,地址偏移量:0x1A
    __IO uint32_t LCKR;         //锁配置寄存器,地址偏移量:0x1C
    __IO uint32_t AFR[2];       //复用功能寄存器,地址偏移量:0x20-0x24
    __IO uint16_t BRR;          //位清除寄存器,地址偏移量:0x28,HD和XL设备有,MD没有
    uint16_t RESERVED3;         //保留,4字节对齐
} GPIO_TypeDef;
```

这样,我们可以用 GPIO_TypeDef 定义一个表示 GPIOx 的结构体变量,通过操作结构体变量对 GPIOx 的寄存器进行配置。

结构体定义中,__IO 的定义为:

```
#define  __IO  volatile
```

每个寄存器变量定义时都要使用 volatile 关键字,保证该寄存器变量存储地址的数据不会被缓存到 Cache 中。

32 位系统中,对于内存的操作以字为单位,导致结构体存在字节对齐问题,RESVEREDn 将 16 位的整型拼成 32 位整型,避免在不同编译器下字操作可能出现的问题,同时将结构体的地址排列和 GPIO 控制器的地址排列一一对应,便于通过指针对 GPIO 寄存器进行操作。

Stm32lxx.h 中对 8 个 GPIO 控制器进行了声明

```
/* !< Peripheral memory map */
#define APB1PERIPH_BASE        PERIPH_BASE
#define APB2PERIPH_BASE        (PERIPH_BASE + 0x10000)
#define AHBPERIPH_BASE         (PERIPH_BASE + 0x20000)
```

```
#define GPIOA_BASE          (AHBPERIPH_BASE + 0x0000)
#define GPIOB_BASE          (AHBPERIPH_BASE + 0x0400)
#define GPIOC_BASE          (AHBPERIPH_BASE + 0x0800)
#define GPIOD_BASE          (AHBPERIPH_BASE + 0x0C00)
#define GPIOE_BASE          (AHBPERIPH_BASE + 0x1000)
#define GPIOH_BASE          (AHBPERIPH_BASE + 0x1400)
#define GPIOF_BASE          (AHBPERIPH_BASE + 0x1800)
#define GPIOG_BASE          (AHBPERIPH_BASE + 0x1C00)
#define GPIOA               ((GPIO_TypeDef * ) GPIOA_BASE)
#define GPIOB               ((GPIO_TypeDef * ) GPIOB_BASE)
#define GPIOC               ((GPIO_TypeDef * ) GPIOC_BASE)
#define GPIOD               ((GPIO_TypeDef * ) GPIOD_BASE)
#define GPIOE               ((GPIO_TypeDef * ) GPIOE_BASE)
#define GPIOH               ((GPIO_TypeDef * ) GPIOH_BASE)
#define GPIOF               ((GPIO_TypeDef * ) GPIOF_BASE)
#define GPIOG               ((GPIO_TypeDef * ) GPIOG_BASE)
```

为便于对用户 GPIO 控制寄存器的配置，ST 提供了 GPIO 标准库函数，头文件位 stm32l1xx_gpio.h，程序源代码位 stm32l1xx_gpio.c。在头文件中，定义了 GPIO_InitTypeDef 结构体用于 I/O 口的初始化配置，该结构体的定义如下：

```
typedef struct
{
    uint32_t GPIO_Pin;                      //需要被配置的 I/O引脚
    GPIOMode_TypeDef GPIO_Mode;             //I/O模式,用于对模式寄存器配置
    GPIOSpeed_TypeDef GPIO_Speed;           //I/O速率,用于对速率寄存器配置
    GPIOOType_TypeDef GPIO_OType;           //输出引脚类型,用于配置输出类型寄存器
    GPIOPuPd_TypeDef GPIO_PuPd;             //上拉下拉类型,用于配置上拉下拉寄存器
}GPIO_InitTypeDef;
```

其中，I/O 模式在 GPIOMode_TypeDef 枚举类型中定义如下：

```
typedef enum
{
    GPIO_Mode_IN   = 0x00,                  //输入模式
    GPIO_Mode_OUT  = 0x01,                  //输出模式
    GPIO_Mode_AF   = 0x02,                  //复用功能
    GPIO_Mode_AN   = 0x03                   //模拟模式
}GPIOMode_TypeDef;
```

GPIO_Pin 的取值定义如下：

```
#define GPIO_Pin_0          ((uint16_t)0x0001)
#define GPIO_Pin_1          ((uint16_t)0x0002)
#define GPIO_Pin_2          ((uint16_t)0x0004)
```

```
#define GPIO_Pin_3                ((uint16_t)0x0008)
#define GPIO_Pin_4                ((uint16_t)0x0010)
#define GPIO_Pin_5                ((uint16_t)0x0020)
#define GPIO_Pin_6                ((uint16_t)0x0040)
#define GPIO_Pin_7                ((uint16_t)0x0080)
#define GPIO_Pin_8                ((uint16_t)0x0100)
#define GPIO_Pin_9                ((uint16_t)0x0200)
#define GPIO_Pin_10               ((uint16_t)0x0400)
#define GPIO_Pin_11               ((uint16_t)0x0800)
#define GPIO_Pin_12               ((uint16_t)0x1000)
#define GPIO_Pin_13               ((uint16_t)0x2000)
#define GPIO_Pin_14               ((uint16_t)0x4000)
#define GPIO_Pin_15               ((uint16_t)0x8000)
#define GPIO_Pin_All              ((uint16_t)0xFFFF)
```

输出模式配置定义如下：

```
typedef enum
{  GPIO_OType_PP = 0x00,             //推挽输出
   GPIO_OType_OD = 0x01              //开漏输出
}GPIOOType_TypeDef;
```

I/O 速率的配置定义如下：

```
typedef enum
{
   GPIO_Speed_400kHz = 0x00,         //极低速
   GPIO_Speed_2MHz   = 0x01,         //低速
   GPIO_Speed_10MHz  = 0x02,         //中等速度
   GPIO_Speed_40MHz  = 0x03          //高速
}GPIOSpeed_TypeDef;
```

上拉和下拉配置定义如下：

```
typedef enum
{  GPIO_PuPd_NOPULL = 0x00,          //浮空
   GPIO_PuPd_UP     = 0x01,          //上拉
   GPIO_PuPd_DOWN   = 0x02           //下拉
}GPIOPuPd_TypeDef;
```

因此，我们对 GPIOx 的配置可以通过如下的程序实现。

```
#define GPIOA   ((GPIO_TypeDef * ) GPIOA_BASE)
GPIOA->MODER = GPIO_Mode_IN;
GPIOA->OSPEEDER = GPIO_Speed_40MHz;
```

ST 提供了 GPIO 操作的函数库如表 5-4 所示。

表 5-4　GPIO 库函数列表

函　数　名	描　　述
GPIO_DeInit	将外设 GPIOx 寄存器重设为默认值
GPIO_Init	根据 GPIO_InitStruct 中指定的参数初始化外设 GPIOx 寄存器
GPIO_StructInit	把 GPIO_InitStruct 中的每一个参数按默认值填入
GPIO_ReadInputDataBit	读取指定端口引脚的输入
GPIO_ReadInputData	读取指定的 GPIO 端口输入
GPIO_ReadOutputDataBit	读取指定端口引脚的输出
GPIO_ReadOutputData	读取指定的 GPIO 端口输出
GPIO_SetBits	设置指定的数据端口位
GPIO_ResetBits	清除指定的数据端口位
GPIO_WriteBit	设置或者清除指定的数据端口位
GPIO_Write	向指定 GPIO 数据端口写入数据
GPIO_PinLockConfig	锁定 GPIO 引脚设置寄存器
GPIO_PinAFConfig	选择 GPIO 引脚的复用功能
GPIO_ToggleBits	翻转 GPIO 引脚的电平状态

1. GPIO_DeInit 函数

函数功能：将 GPIOx 的所有寄存器复位，寄存器值为默认值。

函数原型：void GPIO_DeInit (GPIO_TypeDef * GPIOx)。

输入参数：GPIOx，需要复位的端口号，取值为 GPIOA、GPIOB、GPIOC、…、GPIOH。

应用示例：

```
GPIO_DeInit(GPIOA);                    //将 GPIOA端口复位
```

2. GPIO_Init 函数

函数功能：根据参数 GPIO_InitStruct 初始化 GPIOx 的寄存器。

函数原型：void GPIO_Init(GPIO_TypeDef * GPIOx, GPIO_InitTypeDef * GPIO_InitStruct)。

输入参数 1：GPIOx，需要配置的 I/O 端口，取值为 GPIOA、GPIOB、…、GPIOH。

输入参数 2：GPIO_InitStruct，指向已初始化成员的 GPIO_InitTypeDef 结构体变量的指针。

应用示例：

```
GPIO_InitTypeDef GPIO_InitStructure;
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_All;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
```

```
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;
GPIO_Init(GPIOA, &GPIO_InitStructure);
```

3. GPIO_PinAFConfig 函数

函数功能：设置 I/O 端口的复选功能。

函数原型：void GPIO_PinAFConfig (GPIO_TypeDef * GPIOx, uint16_t GPIO_PinSource, uint8_tGPIO_AF)。

输入参数 1：GPIOx,需要修改的 I/O 端口,取值为 GPIOA、GPIOB、…、GPIOH。

输入参数 2：GPIO_PinSource,端口的 I/O 引脚,取值为 GPIO_PinSourcex,x=0～15,其定义如下：

```
#define GPIO_PinSource0          ((uint8_t)0x00)
#define GPIO_PinSource1          ((uint8_t)0x01)
#define GPIO_PinSource2          ((uint8_t)0x02)
#define GPIO_PinSource3          ((uint8_t)0x03)
#define GPIO_PinSource4          ((uint8_t)0x04)
#define GPIO_PinSource5          ((uint8_t)0x05)
#define GPIO_PinSource6          ((uint8_t)0x06)
#define GPIO_PinSource7          ((uint8_t)0x07)
#define GPIO_PinSource8          ((uint8_t)0x08)
#define GPIO_PinSource9          ((uint8_t)0x09)
#define GPIO_PinSource10         ((uint8_t)0x0A)
#define GPIO_PinSource11         ((uint8_t)0x0B)
#define GPIO_PinSource12         ((uint8_t)0x0C)
#define GPIO_PinSource13         ((uint8_t)0x0D)
#define GPIO_PinSource14         ((uint8_t)0x0E)
#define GPIO_PinSource15         ((uint8_t)0x0F)
```

GPIO_AFSelection：复选功能选择,其取值范围为：

```
//AF 0 selection
#define GPIO_AF_RTC_50Hz       ((uint8_t)0x00)   //RTC 50/60 Hz
#define GPIO_AF_MCO            ((uint8_t)0x00)   //MCO
#define GPIO_AF_RTC_AF1        ((uint8_t)0x00)   //RTC_AF1
#define GPIO_AF_WKUP           ((uint8_t)0x00)   //Wakeup (WKUP1, WKUP2, WKUP3)
#define GPIO_AF_SWJ            ((uint8_t)0x00)   // SW、JTAG
#define GPIO_AF_TRACE          ((uint8_t)0x00)   //TRACE
//AF 1 selection
#define GPIO_AF_TIM2           ((uint8_t)0x01)   //TIM2
//AF 2 selection
#define GPIO_AF_TIM3           ((uint8_t)0x02)   //TIM3
#define GPIO_AF_TIM4           ((uint8_t)0x02)   //TIM4
#define GPIO_AF_TIM5           ((uint8_t)0x02)   // TIM5
//AF 3 selection
```

```
#define GPIO_AF_TIM9            ((uint8_t)0x03)   //TIM9
#define GPIO_AF_TIM10           ((uint8_t)0x03)   //TIM10
#define GPIO_AF_TIM11           ((uint8_t)0x03)   //TIM11
//AF 4 selection
#define GPIO_AF_I2C1            ((uint8_t)0x04)   //I2C1
#define GPIO_AF_I2C2            ((uint8_t)0x04)   // I2C2
//AF 5 selection
#define GPIO_AF_SPI1            ((uint8_t)0x05)   //SPI1
#define GPIO_AF_SPI2            ((uint8_t)0x05)   //SPI2
//AF 6 selection
#define GPIO_AF_SPI3            ((uint8_t)0x06)   //SPI3
//AF 7 selection
#define GPIO_AF_USART1          ((uint8_t)0x07)   // USART1
#define GPIO_AF_USART2          ((uint8_t)0x07)   // USART2
#define GPIO_AF_USART3          ((uint8_t)0x07)   // USART3
//AF 8 selection
#define GPIO_AF_UART4           ((uint8_t)0x08)   //UART4
#define GPIO_AF_UART5           ((uint8_t)0x08)   //UART5
//AF 10 selection
#define GPIO_AF_USB             ((uint8_t)0xA)    // USB Full speed device
//AF 11 selection
#define GPIO_AF_LCD             ((uint8_t)0x0B)   // LCD
//AF 12 selection
#define GPIO_AF_FSMC            ((uint8_t)0x0C)   // FSMC
#define GPIO_AF_SDIO            ((uint8_t)0x0C)   // SDIO
//AF 14 selection
#define GPIO_AF_RI              ((uint8_t)0x0E)   // RI
//AF 15 selection
#define GPIO_AF_EVENTOUT        ((uint8_t)0x0F)   // EVENTOUT
```

应用示例：

```
GPIO_PinAFConfig(GPIOA, GPIO_PinSource9, GPIO_AF_USART1);
```

4. GPIO_PinLockConfig 函数

函数功能：配置 I/O 锁定寄存器。

函数原型：void GPIO_PinLockConfig(GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin)。

输入参数 1：GPIOx，需要配置的 I/O 端口，取值为 GPIOA、GPIOB、…、GPIOH。

输入参数 2：GPIO_Pin，需要锁定的引脚，取值为 GPIO_Pin_x，x=0～15，可以是多个引脚，GPIO_Pin 的每一位表示一个引脚号。

应用示例：

```
GPIO_PinLockConfig(GPIOA, GPIO_Pin_0 | GPIO_Pin_1);
```

5. GPIO_ReadInputData 函数

函数功能：读取指定 I/O 端口的输入数据。

函数原型：uint16_t GPIO_ReadInputData (GPIO_TypeDef * GPIOx)。

输入参数：GPIOx，所要读取的 I/O 端口，取值为 GPIOA、GPIOB、…、GPIOH。

返回值：GPIOx 的输入数据寄存器值。

应用示例：

```
uint16_t ReadValue;
ReadValue = GPIO_ReadInputData(GPIOC);
```

6. GPIO_ReadInputDataBit 函数

函数功能：读取端口指定引脚的输入数据。

函数原型：uint8_t GPIO_ReadInputDataBit (GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin)。

输入参数 1：GPIOx，所要读取的 I/O 端口，取值为 GPIOA、GPIOB、…、GPIOH。

输入参数 2：GPIO_Pin，所要读取的 I/O 端口引脚号，取值为 GPIO_Pin_x，x=0～15。

返回值：该 I/O 引脚的输入数据，为 0 或 1。

应用示例：

```
uint8_t ReadValue;
ReadValue = GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7);
```

7. GPIO_ReadOutputData 函数

函数功能：读取指定 I/O 端口的输出数据寄存器值。

函数原型：uint16_t GPIO_ReadOutputData (GPIO_TypeDef * GPIOx)。

输入参数：GPIOx，所要读取的 I/O 端口，取值为 GPIOA、GPIOB、…、GPIOH。

返回值：该 I/O 端口的输出寄存器值。

应用示例：

```
uint16_t ReadValue;
ReadValue = GPIO_ReadOutputData(GPIOC);
```

8. GPIO_ReadOutputDataBit 函数

函数功能：读取指定端口指定引脚的输出数据值。

函数原型：uint8_t GPIO_ReadOutputDataBit(GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin)。

输入参数 1：GPIOx，所要读取的 I/O 端口，取值为 GPIOA、GPIOB、…、GPIOH。

输入参数 2：GPIO_Pin，所要读取的指定引脚号，取值为 GPIO_Pin_x，x=0～15。

返回值：该引脚的输出数据值，为 0 或 1。

应用示例：

```
uint8_t ReadValue;
```

```
ReadValue = GPIO_ReadOutputDataBit(GPIOB, GPIO_Pin_7);
```

9. GPIO_ResetBits 函数

函数功能：清除指定 I/O 端口的指定 I/O 引脚数据。

函数原型：void GPIO_ResetBits (GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin)。

输入参数 1：GPIOx,所要清除的 I/O 端口,取值为 GPIOA、GPIOB、…、GPIOH。

输入参数 2：GPIO_Pin,所要清除的 I/O 引脚,取值为 GPIO_Pin_x,x=0～15,可以是多个引脚,GPIO_Pin 的每位表示一个引脚号。

应用示例：

```
GPIO_ResetBits(GPIOA, GPIO_Pin_10 | GPIO_Pin_15);
```

10. GPIO_SetBits 函数

函数功能：设置指定 I/O 端口的指定 I/O 引脚数据。

函数原型：void GPIO_SetBits(GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin)。

输入参数 1：GPIOx,所要设置的 I/O 端口,取值为 GPIOA、GPIOB、…、GPIOH。

输入参数 2：GPIO_Pin,所要设置的 I/O 引脚,取值为 GPIO_Pin_x,x=0～15,可以是多个引脚,GPIO_Pin 的每位表示一个引脚号。

应用示例：

```
GPIO_SetBits(GPIOA, GPIO_Pin_10 | GPIO_Pin_15);
```

11. GPIO_StructInit 函数

函数功能：将 GPIO_InitTypeDef 结构体变量的每个成员初始化为默认值。

函数原型：void GPIO_StructInit(GPIO_InitTypeDef * GPIO_InitStruct)。

输入参数：GPIO_InitStruct,所要初始化的 GPIO_InitTypeDef 结构体指针。

调用函数后,输入参数结构体指针的值为：

```
GPIO_Pin   = GPIO_Pin_All;
GPIO_Mode = GPIO_Mode_IN;
GPIO_Speed = GPIO_Speed_400kHz;
GPIO_OType = GPIO_OType_PP;
GPIO_PuPd = GPIO_PuPd_NOPULL;
```

应用示例：

```
GPIO_InitTypeDef GPIO_InitStructure;
GPIO_StructInit(&GPIO_InitStructure);
```

12. GPIO_ToggleBits 函数

函数功能：将指定端口的 I/O 引脚数据翻转。

函数原型：void GPIO_ToggleBits(GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin)。

输入参数 1：GPIOx,所指定的 I/O 端口,取值为 GPIOA、GPIOB、…、GPIOH。

输入参数 2：GPIO_Pin,指定的 I/O 引脚,取值为 GPIO_Pin_x,x=0～15。

应用示例：

```
GPIO_ToggleBits(GPIOA, GPIO_Pin_5);
```

13. GPIO_Write 函数

函数功能：为端口数据寄存器写入指定值。

函数原型：void GPIO_Write(GPIO_TypeDef * GPIOx，uint16_t PortVal)。

输入参数 1：GPIOx，需要写入的指定端口，取值为 GPIOA、GPIOB、…、GPIOH。

输入参数 2：PortVal，需要写入到端口输出数据寄存器的值。

应用示例：

```
GPIO_Write(GPIOA, 0x1101);
```

14. GPIO_WriteBit 函数

函数功能：设置或清除 I/O 端口的一个引脚数据。

函数原型：void GPIO_WriteBit(GPIO_TypeDef * GPIOx，uint16_tGPIO_Pin，BitAction BitVal)。

输入参数 1：GPIOx，需要写入的 I/O 端口，取值为 GPIOA、GPIOB、…、GPIOH。

输入参数 2：GPIO_Pin，需要配置的制定引脚，取值为 GPIO_Pin_x，x=0～15。

输入参数 3：BitVal，指定引脚的值，取值范围为：Bit_RESET 表示引脚置 0，Bit_SET 表示引脚置 1。

应用示例：

```
GPIO_WriteBit(GPIOA, GPIO_Pin_15, Bit_SET);
```

5.4 GPIO 实例

5.4.1 GPIO 寄存器基本操作

我们可以通过 GPIO 控制器的寄存器对 I/O 进行配置。在操作 GPIO 的寄存器之前，首先要开启 GPIO 控制器的时钟，可以通过调用 RCC_AHBPeriphClockCmd (RCC_AHBPeriph_GPIOx，ENABLE)来实现，其中 RCC_AHBPeriph_GPIOx 中的 x 为要使用的 GPIO 端口号。下面以 GPIO_ReadInputData 函数、GPIO_WriteBit 函数和 GPIO_Init 函数的实现和为例阐述如何对 GPIO 的寄存器进行操作。

【例 5-1】 读输入寄存器的寄存器实现。

```
uint16_t GPIO_ReadInputData(GPIO_TypeDef * GPIOx)
{
    //直接返回 IDR 寄存器的值
    return ((uint16_t)GPIOx->IDR);
```

```
}
```

【例 5-2】 配置一个引脚的输出值的寄存器实现。

```
void GPIO_WriteBit(GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin, BitAction BitVal)
{   //Bit_RESET = 0
    if (BitVal != Bit_RESET)
    {   //直接操作 BSRRL将输出数据寄存器对应位置 1
        GPIOx->BSRRL = GPIO_Pin;
    }
    else
    {   //通过 BSRRH将输出数据寄存器对应位清 0
        GPIOx->BSRRH = GPIO_Pin ;
    }
}
```

【例 5-3】 GPIO 寄存器初始化的寄存器实现。

```
void GPIO_Init(GPIO_TypeDef * GPIOx, GPIO_InitTypeDef * GPIO_InitStruct)
{
    uint32_t pinpos = 0x00, pos = 0x00, currentpin = 0x00;
    for (pinpos = 0x00; pinpos < 16; pinpos++)
    {   //对 16个引脚进行配置
        pos = ((uint32_t)0x01) << pinpos;
        //判断是否为输入参数结构体中的引脚
        currentpin = (GPIO_InitStruct->GPIO_Pin) & pos;
        if (currentpin == pos)
        {   //如果是输入参数中的引脚,进行其他参数配置,先将 I/O引脚的模式寄存器 2bit清零
            GPIOx->MODER  &= ~ (GPIO_MODER_MODER0 << (pinpos * 2));
            //然后写入输入参数中的配置
            GPIOx->MODER |= (((uint32_t)GPIO_InitStruct->GPIO_Mode) << (pinpos * 2));
            //如果是输出模式或者复用功能,则需要设置输出速度和输出类型
            if ((GPIO_InitStruct->GPIO_Mode == GPIO_Mode_OUT) ||
                (GPIO_InitStruct->GPIO_Mode == GPIO_Mode_AF))
            {
              //将输出速度寄存器对应的 2个 bit清 0
              GPIOx->OSPEEDR &= ~ (GPIO_OSPEEDER_OSPEEDR0 << (pinpos * 2));
              //写入新的输出速度
              GPIOx->OSPEEDR |= ((uint32_t) (GPIO_InitStruct->GPIO_Speed) << (pinpos * 2));
              //将输出类型寄存器的 1bit清 0
              GPIOx->OTYPER  &= ~ ((GPIO_OTYPER_OT_0) << ((uint16_t)pinpos)) ;
              //写入新的输出类型配置
              GPIOx->OTYPER|= (uint16_t) (((uint16_t)GPIO_InitStruct->GPIO_OType)
                                          << ((uint16_t)pinpos));
            }
```

```
            //上拉下拉寄存器的 2bit 清 0,然后将新的上拉下拉参数写入
            GPIOx->PUPDR &= ~ (GPIO_PUPDR_PUPDR0 << ((uint16_t)pinpos * 2));
            GPIOx->PUPDR |= (((uint32_t)GPIO_InitStruct->GPIO_PuPd) << (pinpos * 2));
        }
    }
}
```

程序中用到的 GPIO_PUPDR_PUPDR0、GPIO_OTYPER_OT_0、GPIO_MODER_MODER0、GPIO_OSPEEDER_OSPEEDR0 的宏定义在 stml1xx.h 中,其表示的是引脚 0 的 PUPDR、OTYPER、MODER 和 OSPEEDR 的配置域在寄存器中的位置。

```
#define GPIO_OSPEEDER_OSPEEDR0        ((uint32_t)0x00000003)
#define GPIO_OTYPER_OT_0              ((uint32_t)0x00000001)
```

5.4.2 GPIO LED 灯控制

【例 5-4】 在 STM32L152-Discovery 开发板上,通过连接到 LED 的 GPIO 引脚对绿色 LED3 灯实现反转控制。LED3 的控制引脚为 PB7,因此我们需要对 GPIOB 端口的第 7 个引脚进行配置,根据电路图,PB7 输出为 1 时灯亮,输出 0 时灯灭。

主程序代码如下:

```
#include "stm32l1xx.h"
#include "stm32L1xx_gpio.h"
#define GREEN_LED GPIO_Pin_7
#define BSRR_VAL 0x80
GPIO_InitTypeDef          GPIO_InitStructure;
void Delay(__IO uint32_t nCount)
{
    while(nCount--)
    {
    }
}
int main(void)
{
    /* GPIOD Periph clock enable */
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOB, ENABLE);
    /* Configure PB7 in output pushpull mode */
    GPIO_InitStructure.GPIO_Pin = GREEN_LED;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_40MHz;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
```

```
        GPIO_Init(GPIOB, &GPIO_InitStructure);

        while (1)
        {
            /* Set PB7 */
            GPIOB->BSRRL = BSRR_VAL;
            Delay(1000);
            /* Reset PB7 */
            GPIOB->BSRRH = BSRR_VAL;
            Delay(1000);
        }
    }
```

程序的执行流程为：

(1) 首先定义两个宏 GREEN_LED 和 BSRR_VAL，其中 GPIO_Pin_7 为系统定义的宏变量，其值为 0x80；BSRR_VAL 用于对 BSRR 寄存器进行赋值，由于需要对 GPIO_Pin_7 进行设置和清除操作，因此 BSRR_VAL 的值设为 0x80。

(2) 用 GPIO_InitTypeDef 结构体类型定义一个 GPIO 寄存器结构体变量 GPIO_InitStructure，用于对 GPIOB 进行初始化。

(3) main 函数中，对结构体变量进行初始化，为 GPIOB 的第 7 个引脚设置为推挽输出，40MHz 速率，浮空，调用 GPIO_Init 函数对 GPIOB 端口的所有寄存器进行初始化。

(4) 调用 RCC_AHBPeriphClockCmd 使能 GPIOB 的时钟，此时 GPIOB 可以正常工作。

(5) 在 while 循环中，通过对 BRSSL 寄存器的第 7 位写 1 将 GPIOB 的输出数据寄存器第 7 位置 1，灯亮。

(6) 调用 delay 函数延时。

(7) 通过对 BRSSH 寄存器的第 7 位写 1 将 GPIOB 的数据寄存器第 7 位置 0，灯灭，调用 delay 函数，循环执行(5)～(7)。

根据 ST 提供的函数库，我们对 LED3 的亮灯和灭灯也可以通过 GPIO_ToggleBits(GPIOB,GREEN_LED)实现。

5.4.3　GPIO 按键输入

【例 5-5】 通过开发板按键作为输入，当按键按下时，获取按键的值，控制 LED3 变亮，按键弹起时，控制 LED3 变灭。

根据 STM32L152-Discovery 开发板的原理图，按键连接在 PA0 口上，当按键按下时，PA0 为低电平，弹起时为高电平。本例程中需要使用两个 I/O 口，PA0 作为输入，PB7 作为输出，主程序如下：

```
#include "stm32l1xx.h"
```

```
#include "stm32L1xx_gpio.h"
int main(void)
{
    GPIO_Configuration();
    while (1)
    {
        if(GPIO_ReadInputDataBit(GPIOA,GPIO_Pin_0))
            GPIO_SetBits(GPIOB, GPIO_Pin_7);
        else
            GPIO_ResetBits(GPIOB,GPIO_Pin_7);
    }
}
```

GPIO 初始化的函数实现如下：

```
void GPIO_Configuration(void)
{
    //使能 AHP 外设时钟
    RCC_AHBPeriphClockCmd( RCC_AHBPeriph_GPIOA|
                           RCC_AHBPeriph_GPIOB, ENABLE);
    //配置 LED3
    GPIO_InitTypeDef  GPIO_InitStructure;
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_7;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_40MHz;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
    GPIO_Init(GPIOB, &GPIO_InitStructure);
    //配置按键
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
}
```

Chapter 6

第 6 章　异常和中断处理技术

【导读】 中断是计算机管理外设和处理异常的一种重要手段。本章首先介绍中断的基本概念，对 Cortex-M3 处理器的中断向量表给出了简要说明，然后详细分析了 Cortex-M3 处理器的嵌套向量中断控制器和外部中断控制器，介绍了中断向量控制器和外部中断控制器的相应库函数，最后给出通过按键产生中断控制 LED 灯切换显示的实验代码，并对中断处理流程进行了综合分析。

6.1　中断的基本概念

中断是计算机中的一个十分重要的概念，在现代计算机中毫无例外地都要采用中断技术。可以举一个日常生活中的例子来说明中断的概念，假如你正在厨房做饭，电话铃响了，你暂停做饭去接电话。通话完毕再继续做饭。这个例子就表现了中断及其处理过程：电话铃声使你暂时中止当前的做饭工作，而去处理更为急需处理的事情（接电话），把急需处理的事情处理完毕之后，再回头来继续原来的事情。在这个例子中，电话铃声称为“中断请求”，暂停做饭去接电话可以称为“中断响应”，接电话的过程就是“中断处理”。相应地，计算机在执行程序的过程中，由于出现特殊情况，使得 CPU 中止正在运行的程序，而转去执行处理该特殊情况的处理程序。这个特殊情况即为中断，这个处理特殊情况的处理程序即为中断服务程序。当中断服务程序执行完毕后，再继续执行原来的程序。

计算机中采用中断技术主要是为了提高计算机系统的执行效率。图 6-1 给出了采用查询和中断两种外部设备管理方式。对于打印输出操作，CPU 工作速度和传送数据的速度都很快，而打印机打印的速度慢。如果采用查询技术，CPU 将经常处于等待状态，效率极低。而采用了中断方式后，CPU 可以进行其他的工作，只在打印机缓冲区中的当前内容打印完毕发出中断请求之后，才予以响应，暂时中断当前工作转去执行向缓冲区传送数据，传送完成后又返回执行原来的程序。

中断指的是当出现需要时，CPU 暂时停止当前程序的执行转而执行处理新情况的程序和执行过程；这种处理新情况的程序或执行过程，称为中断服务程序 ISR；中断服务程序的入口地址，称为中断向量；中断向量一般存放在计算机内存的某个特定位置，存放中断向量

的存储空间称为中断向量表;存储中断向量的地址,称为中断向量地址。

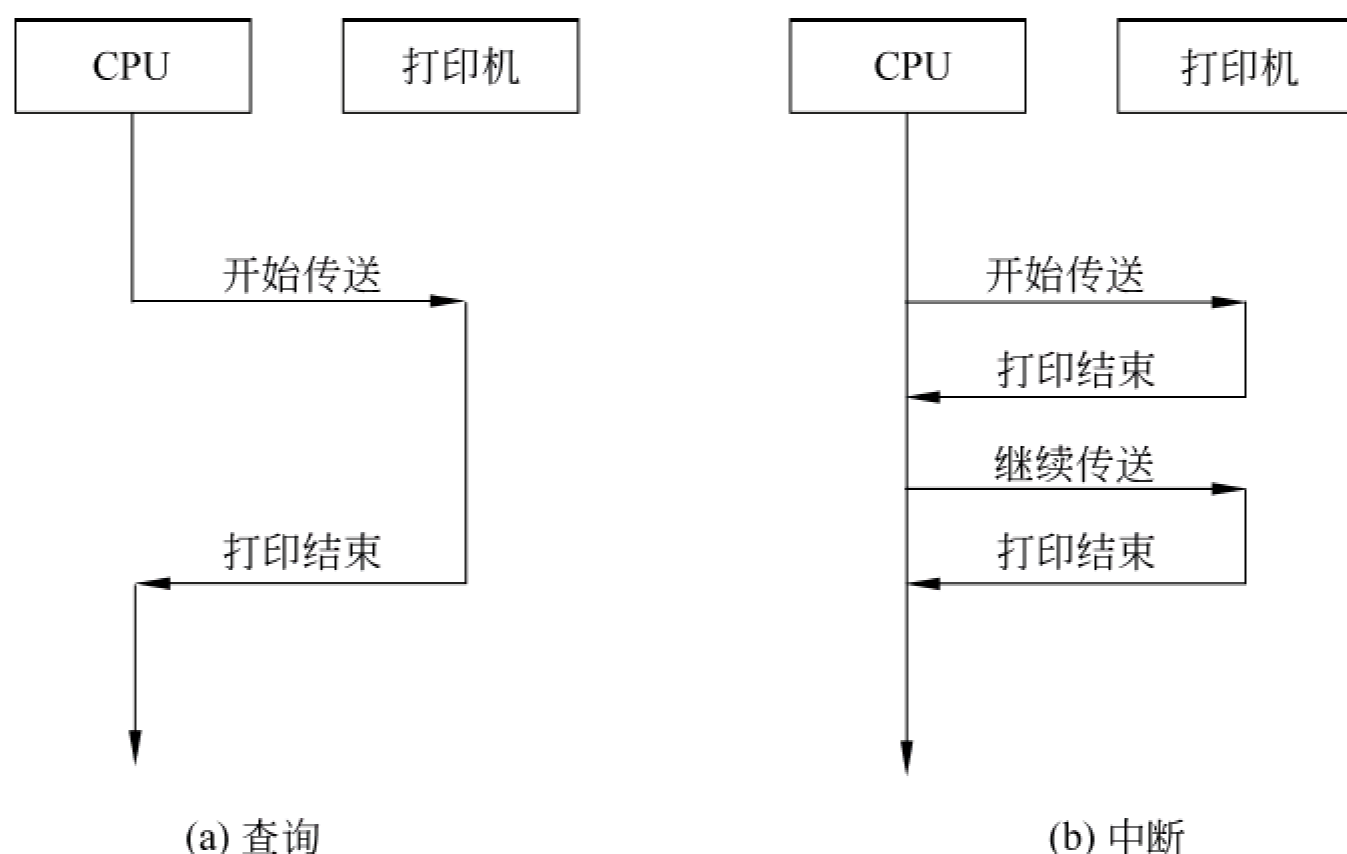


图 6-1　两种外部设备工作方式

6.2　中断向量表

Cortex-M3 将打断 CPU 执行的事件分为异常与中断,其区别在于中断来自于 Cortex 内核外部,比如各种片上外设、外部中断请求等;而异常则是由于 Cortex 内核在执行指令或者访问存储等操作时所产生的。Cortex-M3 支持最多 256 个异常(广义的异常),处理器为了识别异常,对每个异常进行了唯一编号,其中编号 0～15 的为系统异常(狭义的异常),编号 16～255 的为中断,支持多达 240 个外部中断。嵌套向量中断控制器(NVIC)负责 Cortex-M3 中断管理控制,提供可屏蔽的、可嵌套的、具有动态优先级的中断管理。

【思考题:异常和中断有何区别?**】**

STM32L152 处理器的中断向量表如表 6-1 所示,每个中断或异常都有一个唯一的 IRQ 编号,异常的 IRQ 编号均为负数,中断的 IRQ 编号从 0 开始,IRQ 编号和 Cortex-M3 异常编号之间差 16。异常包括:复位 Reset、不可屏蔽中断 NMI、硬件错误 HardFault、内存管理错误 MemManage、总线错误 BusFault、指令错误 UsageFault、特权指令调用 SVCall、挂起调用 PendSV 和系统时钟 SysTick;中断包括看门狗中断,外部 I/O 中断、定时器中断等。每种异常或中断都有优先级,其中复位、NMI 和 HardFault 的优先级是固定的,其他的异常和中断的优先级均可配置,优先级值越小,优先级越高。复位中断优先级最高(−3),中断向量地址为 0x00000004,当用户按下复位键后,不论当前正在执行的是用户程序还是中断服务程序,执行控制都会转到存储在地址 0x00000004 中的程序地址去执行代码。

表 6-1　中断向量表

IRQ 编号	中断/异常编号	优先级次序	优先级类型	名称	说　明	地址
—	0	—	—	—	SP 指针初始值	0x00000000
	1	−3	固定	Reset	复位	0x00000004
−14	2	−2	固定	NMI	不可屏蔽中断	0x00000008
−13	3	−1	固定	HardFault	故障升级	0x0000000C
−12	4	0	可设置	MemManage	存储器管理故障	0x00000010
−11	5	1	可设置	BusFault	总线故障	0x00000014
−10	6	2	可设置	UsageFault	用法错误	0x00000018
—	—				保留	
−5	11	3	可设置	SVCall	SWI 系统调用	0x0000002C
−4	12	4	可设置	DebugMonitor	调试监控器	0x00000030
—	—	—			保留	0x00000034
−2	14	5	可设置	PendSV	可挂起系统服务	0x00000038
−1	15	6	可设置	SysTick	系统嘀嗒定时器	0x0000003C
0	16	7	可设置	WWDG	窗口定时器中断	0x00000040
1	17	8	可设置	PVD	电压检测中断	0x00000044
2	18	9	可设置	TAMPER	入侵检测中断	0x00000048
3	19	10	可设置	RTC	实时时钟中断	0x0000004C
4	20	11	可设置	FLASH	内存全局中断	0x00000050
5	21	12	可设置	RCC	复位和时钟控制中断	0x00000054
6	22	13	可设置	EXT0	EXT0 线中断	0x00000058
7	22	14	可设置	EXT1	EXT1 线中断	0x0000005C
8	24	15	可设置	EXT2	EXT2 线中断	0x00000060
9	25	16	可设置	EXT3	EXT3 线中断	0x00000064
10	26	17	可设置	EXT4	EXT4 线中断	0x00000068
⋮	⋮	⋮	⋮	⋮	⋮	⋮

文件 startup_stm3211xx_md.s 中，使用 DCD 指令将中断向量表填写到存储器中。当有异常或中断产生时，处理器会跳转到 DCD 后面的函数处执行中断服务程序，完整的中断向量表如下。

```
; Vector Table Mapped to Address 0 at Reset
```

```
                AREA    RESET, DATA, READONLY
                EXPORT  __Vectors
                EXPORT  __Vectors_End
                EXPORT  __Vectors_Size

__Vectors       DCD     __initial_sp                ; Top of Stack
                DCD     Reset_Handler               ; Reset Handler
                DCD     NMI_Handler                 ; NMI Handler
                DCD     HardFault_Handler           ; Hard Fault Handler
                DCD     MemManage_Handler           ; MPU Fault Handler
                DCD     BusFault_Handler            ; Bus Fault Handler
                DCD     UsageFault_Handler          ; Usage Fault Handler
                DCD     0                           ; Reserved
                DCD     0                           ; Reserved
                DCD     0                           ; Reserved
                DCD     0                           ; Reserved
                DCD     SVC_Handler                 ; SVCall Handler
                DCD     DebugMon_Handler            ; Debug Monitor Handler
                DCD     0                           ; Reserved
                DCD     PendSV_Handler              ; PendSV Handler
                DCD     SysTick_Handler             ; SysTick Handler
                ; External Interrupts
                DCD     WWDG_IRQHandler             ; Window Watchdog
                DCD     PVD_IRQHandler              ; PVD through EXTI Line detect
                DCD     TAMPER_STAMP_IRQHandler     ;Tamper and TimeStamp
                DCD     RTC_WKUP_IRQHandler         ; RTC Wakeup
                DCD     FLASH_IRQHandler            ; FLASH
                DCD     RCC_IRQHandler              ; RCC
                DCD     EXTI0_IRQHandler            ; EXTI Line 0
                DCD     EXTI1_IRQHandler            ; EXTI Line 1
                DCD     EXTI2_IRQHandler            ; EXTI Line 2
                DCD     EXTI3_IRQHandler            ; EXTI Line 3
                DCD     EXTI4_IRQHandler            ; EXTI Line 4
                DCD     DMA1_Channel1_IRQHandler    ; DMA1 Channel 1
                DCD     DMA1_Channel2_IRQHandler    ; DMA1 Channel 2
                DCD     DMA1_Channel3_IRQHandler    ; DMA1 Channel 3
                DCD     DMA1_Channel4_IRQHandler    ; DMA1 Channel 4
                DCD     DMA1_Channel5_IRQHandler    ; DMA1 Channel 5
                DCD     DMA1_Channel6_IRQHandler    ; DMA1 Channel 6
                DCD     DMA1_Channel7_IRQHandler    ; DMA1 Channel 7
                DCD     ADC1_IRQHandler             ; ADC1
```

```
                DCD     USB_HP_IRQHandler           ; USB High Priority
                DCD     USB_LP_IRQHandler           ; USB Low  Priority
                DCD     DAC_IRQHandler              ; DAC
                DCD     COMP_IRQHandler             ; COMP through EXTI Line
                DCD     EXTI9_5_IRQHandler          ; EXTI Line 9…5
                DCD     LCD_IRQHandler              ; LCD
                DCD     TIM9_IRQHandler             ; TIM9
                DCD     TIM10_IRQHandler            ; TIM10
                DCD     TIM11_IRQHandler            ; TIM11
                DCD     TIM2_IRQHandler             ; TIM2
                DCD     TIM3_IRQHandler             ; TIM3
                DCD     TIM4_IRQHandler             ; TIM4
                DCD     I2C1_EV_IRQHandler          ; I2C1 Event
                DCD     I2C1_ER_IRQHandler          ; I2C1 Error
                DCD     I2C2_EV_IRQHandler          ; I2C2 Event
                DCD     I2C2_ER_IRQHandler          ; I2C2 Error
                DCD     SPI1_IRQHandler             ; SPI1
                DCD     SPI2_IRQHandler             ; SPI2
                DCD     USART1_IRQHandler           ; USART1
                DCD     USART2_IRQHandler           ; USART2
                DCD     USART3_IRQHandler           ; USART3
                DCD     EXTI15_10_IRQHandler        ; EXTI Line 15…10
                DCD     RTC_Alarm_IRQHandler        ; RTC Alarm through EXTI Line
                DCD     USB_FS_WKUP_IRQHandler      ; USB FS Wakeup from suspend
                DCD     TIM6_IRQHandler             ; TIM6
                DCD     TIM7_IRQHandler             ; TIM7
__Vectors_End
```

文件 startup_stm3211xx_md.s 中，中断向量 Reset_Handler 对应复位中断服务程序代码如下：

```
; Reset handler routine
Reset_Handler   PROC
                EXPORT  Reset_Handler               [WEAK]
                IMPORT  __main
                IMPORT  SystemInit
                LDR     R0, =SystemInit
                BLX     R0
                LDR     R0, =__main
                BX      R0
                ENDP
```

该段代码的功能是调用 SystemInit 来配置时钟，然后进入用户的入口函数 main。

其他中断服务函数的定义在 stm32l1xxx_it. c 中：

```
//以下为系统异常服务程序,异常服务程序的实现一般为空函数或者死循环
void NMI_Handler(void)
{
    /* 不可屏蔽中断 */
}
void HardFault_Handler(void)
{
    while (1) ;     /* 硬件错误异常发生时,执行该函数,死循环 */
}
void MemManage_Handler(void)
{
    while (1) ;     /* 存储器故障异常发生时,执行该函数,死循环 */
}
void SysTick_Handler(void)
{
    /* 系统定时器异常,可以在此添加中断处理代码 */
}
...
//以下为外围控制器中断服务程序,PPP_IRQHandler 要和汇编文件中断向量表中的一致
void PPP_IRQHandler(void)
{
    /* PPP外设的中断服务程序,可以在此添加中断处理代码 */
}
```

6.3 中断的执行过程

6.3.1 中断响应基本流程

1. 处理器模式切换

当没有异常发生时，处理器处在线程模式，当进入中断处理(ISR)或故障处理激活时，处理器将进入异常处理模式，不同类型异常处理所对应的处理器工作模式、访问级别以及栈的使用是有所不同的，也就是激活等级不同。

2. 中断响应过程

中断响应的过程为：

(1) CPU 保护现场，将返回地址、关键寄存器压栈；

(2) 获取异常/中断编码；

(3) 查找中断向量表，获得中断服务程序 ISR 的地址；

（4）执行中断服务程序；

（5）中断返回，回复现场，继续执行。

响应异常的第一个行动，就是自动保存现场的必要部分：依次把 xPSR，PC，LR，R12 以及 R0～R3 由硬件自动压入适当的堆栈中，如图 6-2 所示。如果当响应异常时，当前的代码正在使用 PSP，则压入 PSP，即使用线程堆栈；否则压入 MSP，使用主堆栈。一旦进入了中断服务程序，就将一直使用主堆栈。

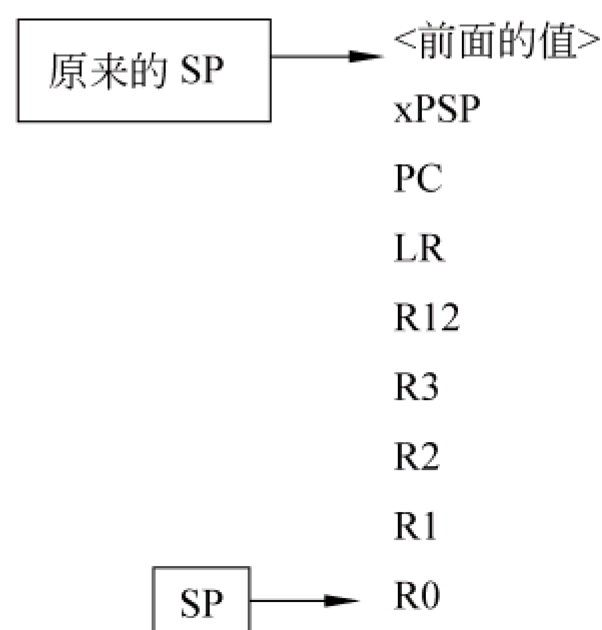


图 6-2　入栈的寄存器及其存储位置

【思考题：为何只保存 R0～R3，其他寄存器要保存吗？】

当数据总线入栈时，指令总线（ICode）同时从向量表中找出正确的异常向量，然后在服务程序的入口处预取指，入栈与取指这两个工作同时进行。

在入栈和取向量的工作都完毕之后，执行服务例程之前，还要更新一系列的寄存器。

- SP：在入栈中会把栈指针（PSP 或 MSP）更新到新的位置。在执行服务例程后，将由 MSP 负责对栈的访问。
- PSR：IPSR 会被更新为新响应的异常编号。
- PC：在向量取出完毕后，PC 将指向服务例程的入口地址。
- LR：LR 不用返回地址，而是被赋为特殊值 EXC_RETURN，在异常返回时使用。

表 6-2 为异常进入的步骤。

表 6-2　异常进入过程

动　　作	描　　述
8 个寄存器压栈	在所选的堆栈上将 xPSR，PC，R0，R1，R2，R3，R12，LR 压栈，复位异常不执行该动作
读向量表	读存储器中的向量表，地址为向量表基址＋（异常号 4）。ICode 总线上的读操作能够与 DCode 总线上的寄存器压栈操作同时执行
从向量表中读 SP	只在复位时执行该动作，将 SP 更新为向量表中栈顶的值。选择堆栈，压栈和出栈之外的其他异常不能修改 SP
更新 P	利用向量表读出的位置更新 PC。直到第一条指令开始执行时，才能处理迟来异常
加载流水线	从向量表指向的位置加载指令。它与寄存器压栈操作同时执行
更新 LR	LR 设置为 EXC_RETURN，以便从异常中退出。EXC_RETURN 参见表 6-3

如表 6-4 所示，当异常服务例程执行完毕后，需要执行“异常返回”动作序列，从而恢复先前的系统状态，返回时需要使用 EXC_RETURN 的值。EXC_RETURN 只有低 4 位有效，其余位必须置为 1，其低四位的取值见表 6-3。

将先前压入栈中的寄存器出栈，栈指针恢复中断前的值。更新 NVIC 寄存器，清除中断激活位等。

表 6-3 EXC_RETURN 的返回值

EXC_RETURN[3：0]	功　能
0b0001	返回处理模式，异常返回，获得来自主堆栈的状态，返回时指令执行使用主栈 MSP
0b1001	返回线程模式，异常返回，获得来自主堆栈的状态，返回时指令执行使用主栈 MSP
0b1101	返回线程模式，异常返回，获得来自进程堆栈的状态，返回时指令执行使用进程栈 PSP

表 6-4 异常返回过程

动　作	描　述
8 个寄存器出栈	如果没有被抢占，则将 PC，xPSR，R0，R1，R2，R3，R12，LR 从所选的堆栈中出栈(堆栈由 EXC_RETURN 选择)，并调整 SP
加载当前激活的中断号	加载来自被压栈的 IPSR 的位[8：0]中的当前激活的中断号。处理器用它来跟踪返回到哪个异常以及返回时清除激活位。当位[8：0]为 0 时，处理器返回线程模式
选择 SP	如果返回到异常，SP 为 SP_main，如果返回到线程模式，则 SP 为 SP_main 或 SP_process

Cortex-M3 支持由软件指定优先级，优先级范围为 0～255，0 优先级最高，255 优先级最低。为了对具有大量中断的系统加强优先级控制，NVIC 支持优先级分组机制，优先级的值分为抢占优先级区和次优先级区。我们将抢占优先级称为组优先级。如果有多个挂起异常共用相同的组优先级，则需使用次优先级区来决定同组中的异常的优先级，这就是同组内的次优先级。组优先级和次优先级的结合就是通常所说的优先级。如果两个挂起异常具有相同的优先级，则挂起异常的编号越低优先级越高。

图 6-3 为 8 位优先级 PRI_N 的占先优先级区(x)和次优先级区(y)的配置。

PRIGROUP[2:0]	中断优先级区，PRI_N[7:0]				
	二进制点的位置	占先区	次优先级区	占先优先级的数目	次优先级的数目
b000	bxxxxxxx.y	[7:1]	[0]	128	2
b001	bxxxxxx.yy	[7:2]	[1:0]	64	4
b010	bxxxxx.yyy	[7:3]	[2:0]	32	8
b011	bxxxx.yyyy	[7:4]	[3:0]	16	16
b100	bxxx.yyyyy	[7:5]	[4:0]	8	32
b101	bxx.yyyyyy	[7:6]	[5:0]	4	64
b110	bx.yyyyyyy	[7]	[6:0]	2	128
b111	b.yyyyyyyy	无	[7:0]	0	256

图 6-3 中断优先级配置

高抢占优先级的中断会打断当前用户程序或当前正在执行的低抢占优先级的中断服务程序，即中断嵌套；在抢占优先级相同的情况下，次优先级高的中断优先被响应。但是，在抢

占优先级相同的情况下,如果有低次优先级的中断服务程序正在执行,则高次优先级的中断不能打断,即不能嵌套中断。当两中个断源的抢占式优先级相同时,这两个中断没有嵌套关系,当一个中断到来后如果另一个中断正在处理,这个后到的中断就要等到前一个中断处理完之后才能被处理。如果这两个中断同时到达,中断控制器则根据它们的响应优先级高低来决定先处理哪一个;如果它们的抢占式优先和次优先级都相等,则根据它们在中断表中的排位顺序决定先处理哪一个。

6.3.2　中断优化技术

为提高中断的响应速度,在优先级的基础上,Cortex-M3 提供了以下中断优化技术。

1. 抢占

当新的异常比当前异常有更高优先级时,则中断当前操作流程,响应新的异常并执行其 ISR,即发生中断嵌套,如图 6-4 所示。

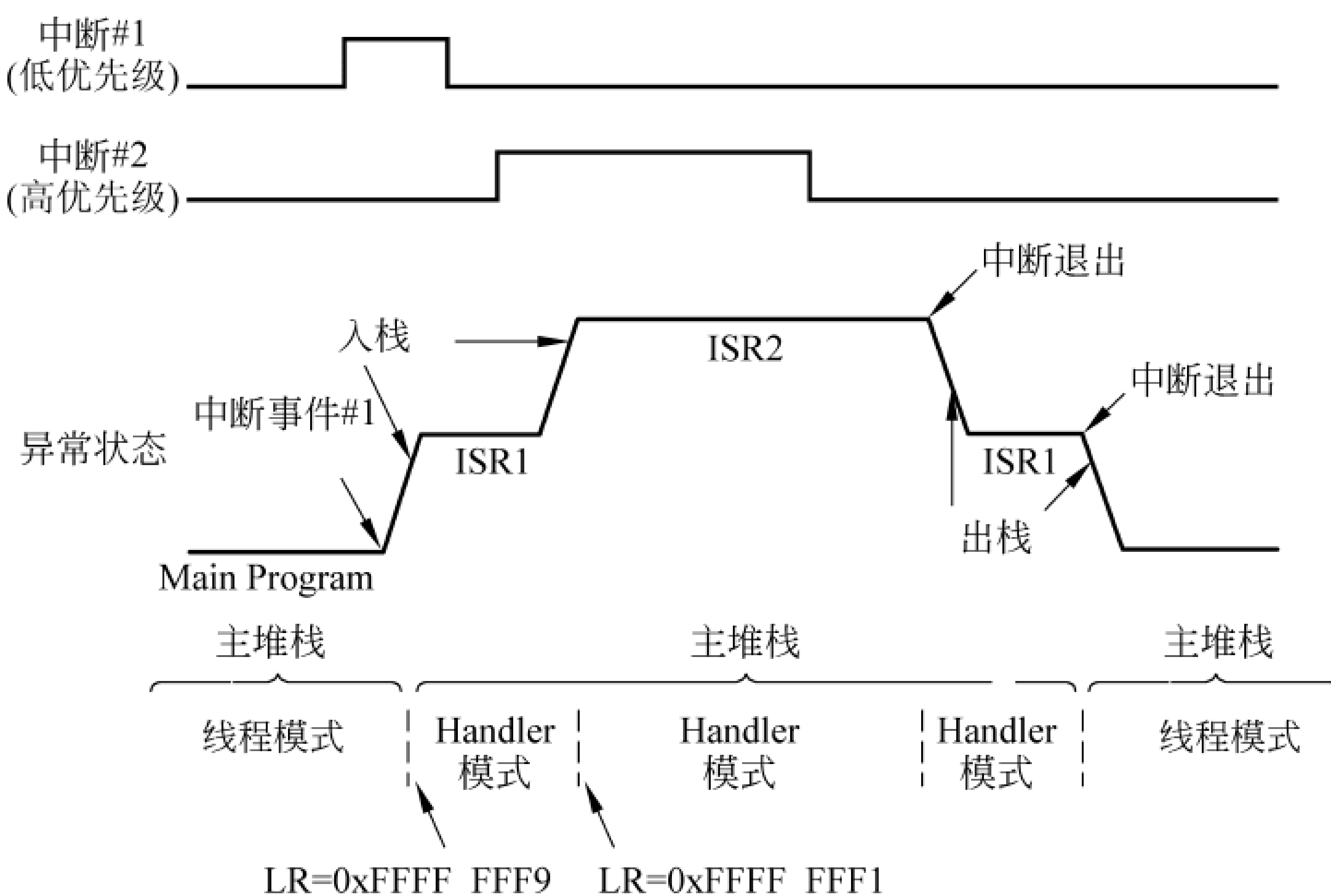


图 6-4　抢占过程

2. 末尾连锁

末尾连锁能够在两个中断之间没有多余的状态保存和恢复指令的情况下实现背对背处理。在退出 ISR 并进入另一个中断时,处理器略过 8 个寄存器的出栈和压栈操作,因为它对堆栈的内容没有影响。如果挂起中断的优先级比所有被压栈的异常的优先级都高,则处理器执行末尾连锁直接取出挂起中断的向量,在退出前一个 ISR 之后六个周期,开始执行被末尾连锁的 ISR,如图 6-5 所示。

3. 返回

如果挂起的异常中没有比栈中的 ISR 异常优先级更高的,则处理器执行返回操作,恢复进入 ISR 之前的状态,在恢复现场的过程中,如果此时有更高优先级的中断到来,但处理

器还没有恢复完成现场，此时放弃恢复现场的过程，直接将更高优先级的中断作为尾链处理。

【思考题：已经弹出栈的值如何处理?】

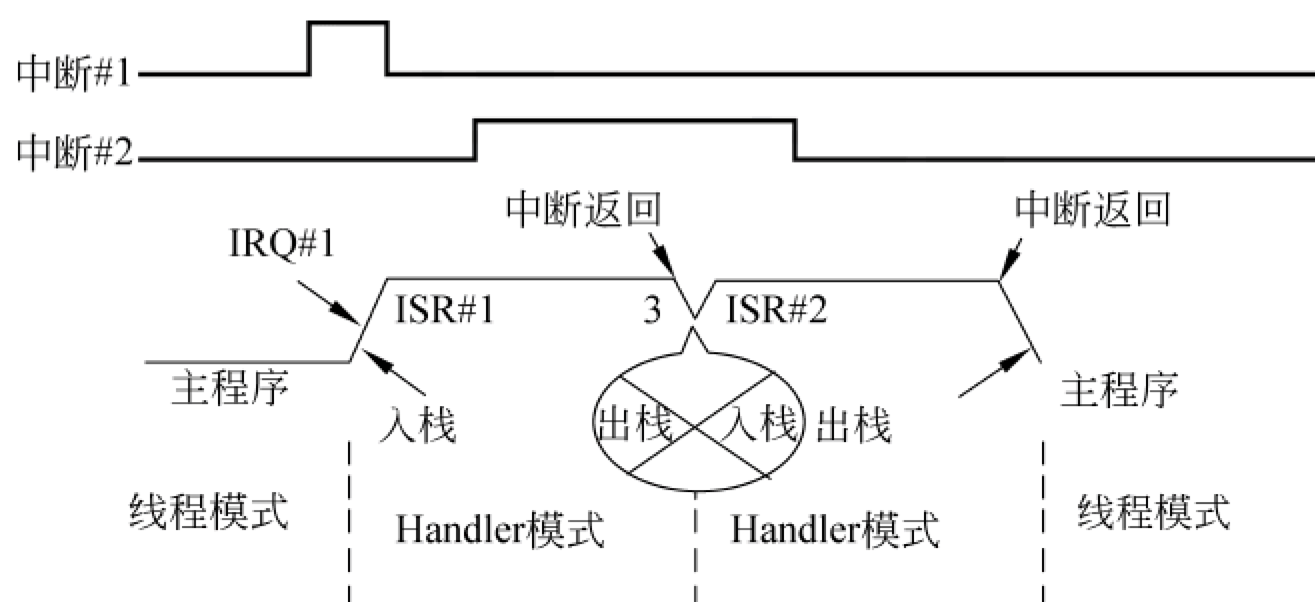


图 6-5　尾链处理过程

4. 迟到

如果前一个 ISR 还没有进入执行阶段，并且迟来中断的优先级比前一个中断的优先级要高，则迟来中断能够抢占前一个中断，如图 6-6 所示。响应迟来中断时需执行新的取向量地址和 ISR 预取操作。迟来中断不保存状态，因为状态保存已经被最初的中断执行过了，因此不需要重复执行。

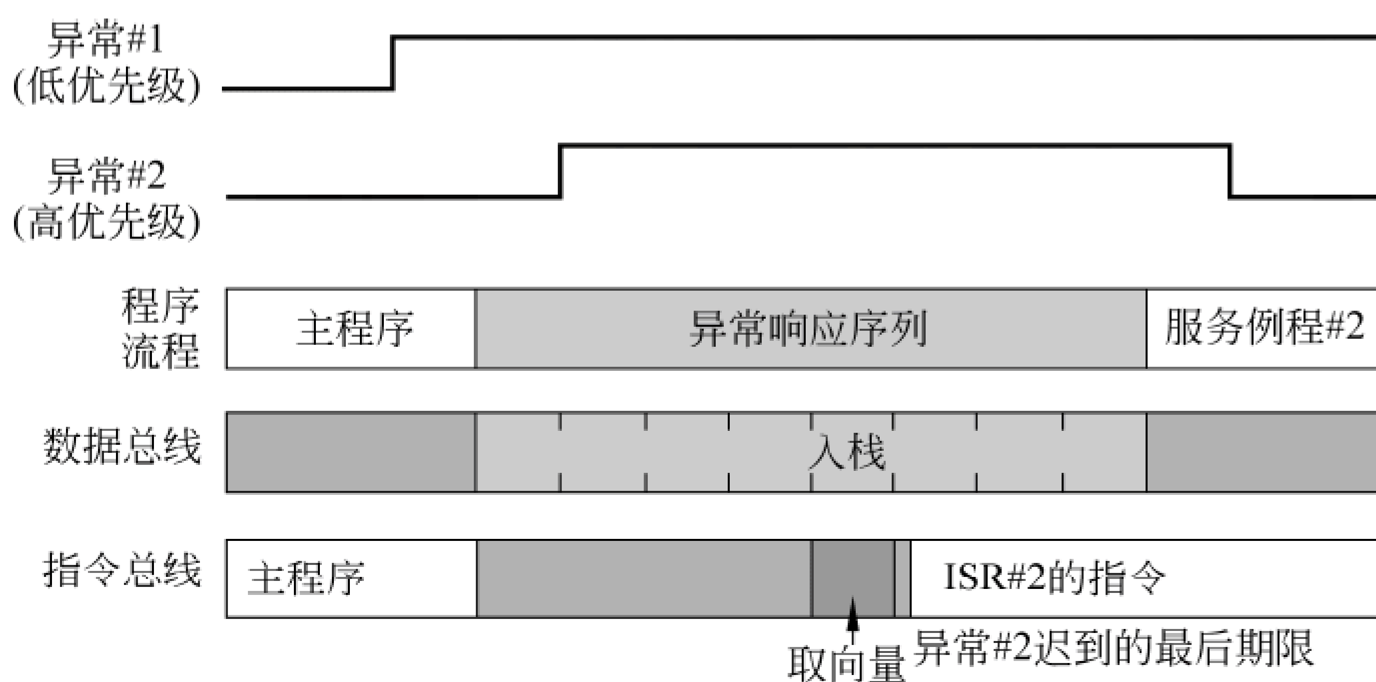


图 6-6　迟到中断响应过程

6.3.3　系统异常

1. 复位

表 6-5 为复位的过程。复位后，CM3 读取下列两个 32 位整数的值。

(1) 从地址 0x0000,0000 处取出 MSP 的初始值。

(2) 从地址 0x0000,0004 处取出 PC 的初始值(复位异常处理函数)，然后从这个值所对应的地址处取指，如图 6-7 所示。

表 6-5　复位动作

动　　作	描　　述
NVIC 复位，内核保持在复位状态	NVIC 对它的大部分寄存器进行清零。处理器位于线程模式，优先级为特权模式，堆栈设置为主堆栈
NVIC 将内核从复位状态释放	NVIC 将内核从复位状态释放
内核设置堆栈	内核从向量表偏移 0 中读取最初的 MSP 值
内核设置 PC 和 LR	内核从向量表偏移中读取最初的 PC。LR 设置为 0xFFFFFFFF
运行复位程序	NVIC 的中断被禁止，NMI 和硬件故障未被禁止



图 6-7　复位异常处理过程

2．故障

故障时处理器内部产生的异常，能够产生中止故障的 4 个事件包括：

(1) 指令取指或向量表加载时的总线错误；

(2) 数据访问时的总线错误；

(3) 内部检测到的错误，例如未定义的指令或试图用 BX 指令来改变状态；

(4) 由于违反了特权模式或未管理的区域而引起的 MPU 故障。

① 存储器故障：触犯了 MPU 设置的保护规范和某些非法访问引起的存储器错误，例如访问了所有 MPU 覆盖范围之外的地址，访问了没有存储器与之对应的空地址，往只读区写数据，用户级下访问了只允许在特权级下访问的地址等。

② 总线故障：当 AHB 接口上正在传送数据时，如果回复了一个错误信号，则会产生总线故障，例如取指过程中的预取流产，数据读写过程中的数据流产等。

③ 用法故障：错误使用导致的故障，如执行了协处理器指令(Cortex-M3 没有协处理器)、执行了未定义的指令、尝试进入 ARM 状态、无效的中断返回(LR 中包含了无效/错误的值)、使用多重加载/存储指令时，地址没有对齐等。

④ 硬件故障：上述常规故障处理中发生问题时，故障升级成为硬件故障。

3．SVC 系统调用和 PendSV 挂起调用

SVC(系统服务调用)和 PendSV(挂起系统调用)异常多用于在操作系统。SVC 用于产生系统函数的调用请求，从用户级切入到特权级，如用户程序使用 SVC 发出对系统服务函数的呼叫请求，以这种方法调用它们来间接访问硬件。PendSV 可以像普通的中断一样被悬起的，可以让其他重要的任务完成后才执行该异常处理，PendSV 的典型使用场合是操作系统的上下文切换时(在不同任务之间切换)。

4．SysTick 定时器

SysTick 定时器被捆绑在 NVIC 中，用于产生 SYSTICK 异常。大多操作系统需要一个

硬件定时器来产生操作系统需要的滴答中断,作为整个系统的时基,以维持操作系统进程调度,Cortex-M3 处理器内部包含了一个简单的定时器。所有的 Cortex-M3 芯片都带有 SysTiuck 定时器这样使得软件在不同 Cortex-M3 器件间的移植工作得以化简。

6.4 嵌套向量中断控制器 NVIC

6.4.1 STM32L152 NVIC

Cortex-M3 处理器中定义了 8 个位来设置中断源的优先级,STM32L152 实际只使用了其中的高 4 位,低 4 位置为 0,如表 6-6 所示。分别有 0～4 共 5 组中断优先级方式。例如,对于分组 3 的方式,抢占优先级占据 3 位,则一共有 0～7 共 8 个抢占优先级,而从优先级只占 1 位,则一共只有 0 和 1 共两个次优先级。

表 6-6 STM32L152 中断优先级分组

分　组	抢占优先级位数和取值范围	从优先级位数和取值范围
0	0(无)	4(0～15)
1	1(0～1)	3(0～7)
2	2(0～3)	2(0～3)
3	3(0～7)	1(0～1)
4	4(0～15)	0(无)

STM32L152 在实现 NVIC 时,根据具体情况对 Cortex-M3 的 NVIC 进行了配置,其特点是:最多支持 81 个外部中断(Cortex-M3 支持 240 个),16 种优先级。

【思考题:如果 STM32L152 配置分组为 1,抢占优先级为 1,从优先级为 4,那么优先级的值为多少?**】**

6.4.2 NVIC 寄存器

NVIC 控制器包含在 Cortex-M3 内核中,除了中断和系统异常控制外,还用来实现系统控制。其寄存器列表如表 6-7 所示。

表 6-7 NVIC 寄存器

名　称	类型	地址	复位值
中断控制类型寄存器	只读	0xE000E004	由配置定义
IRQ0～IRQ31 使能设置寄存器	读写	0xE000E100	0x00000000

续表

名　　称	类型	地址	复位值
⋮	⋮	⋮	⋮
IRQ224～IRQ239 使能设置寄存器	读写	0xE000E11C	0x00000000
IRQ0～IRQ31 使能清除寄存器	读写	0xE000E180	0x00000000
⋮	⋮	⋮	⋮
IRQ224～IRQ239 使能清除寄存器	读写	0xE000E19C	0x00000000
IRQ0～IRQ31 挂起设置寄存器	读写	0xE000E200	0x00000000
⋮	⋮	⋮	⋮
IRQ224～IRQ239 挂起设置寄存器	读写	0xE000E21C	0x00000000
IRQ0～IRQ31 挂起清除寄存器	读写	0xE000E280	0x00000000
⋮	⋮	⋮	⋮
IRQ224～IRQ239 挂起清除寄存器	读写	0xE000E29C	0x00000000
IRQ0～IRQ31 激活位寄存器	只读	0xE000E29C	0x00000000
⋮	⋮	⋮	⋮
IRQ224～IRQ239 激活位寄存器	只读	0xE000E31C	0x00000000
IRQ0～IRQ31 优先级寄存器	读写	0xE000E400	0x00000000
⋮	⋮	⋮	⋮
IRQ236～IRQ239 优先级寄存器	读写	0xE000E4F0	0x00000000
中断控制状态寄存器	读写或只读	0xE000ED04	0x00000000
向量表偏移寄存器	读写	0xE000ED08	0x00000000
应用中断/复位控制寄存器	读写	0xE000ED0C	0x00000000
系统控制寄存器	读写	0xE000ED10	0x00000000
配置控制寄存器	读写	0xE000ED14	0x00000000
系统处理器 4～7 优先级寄存器	读写	0xE000ED18	0x00000000
系统处理器 8～11 优先级寄存器	读写	0xE000ED1C	0x00000000
系统处理器 12～15 优先级寄存器	读写	0xE000ED20	0x00000000
系统处理器控制与状态寄存器	读写	0xE000ED24	0x00000000
可配置故障状态寄存器	读写	0xE000ED28	0x00000000
硬故障状态寄存器	读写	0xE000ED2C	0x00000000
调试故障状态寄存器	读写	0xE000ED30	0x00000000
存储器管理地址寄存器	读写	0xE000ED34	不可预测

续表

名　　称	类型	地址	复位值
总线故障地址寄存器	读写	0xE000ED38	不可预测
PFR0 处理器功能寄存器 0	只读	0xE000ED40	0x00000000
PFR1 处理器功能寄存器 1	只读	0xE000ED44	0x00000000
DFR0 调试功能寄存器 0	只读	0xE000ED48	0x00000000
AFR0 辅助功能寄存器 0	只读	0xE000ED4C	0x00000000
MMFR0 存储器模型功能寄存器 0	只读	0xE000ED50	0x00000000
MMFR1 存储器模型功能寄存器 1	只读	0xE000ED54	0x00000000
MMFR2 存储器模型功能寄存器 2	只读	0xE000ED58	0x00000000
MMFR3 存储器模型功能寄存器 3	只读	0xE000ED5C	0x00000000
ISAR0 ISA 功能寄存器 0	只读	0xE000ED60	0x01141110
ISAR1 ISA 功能寄存器 1	只读	0xE000ED64	0x02111000
ISAR2 ISA 功能寄存器 2	只读	0xE000ED68	0x21112231
ISAR3 ISA 功能寄存器 3	只读	0xE000ED6C	0x01111110
ISAR4 ISA 功能寄存器 4	只读	0xE000ED70	0x01310102
软件触发中断寄存器	只写	0xE000EF00	—

在 NVIC 异常中断控制中，我们主要关注 Cortex-M3 外部的 240 个中断，这些中断的配置主要包括 5 类寄存器：中断使能寄存器 ISER、中断清除寄存器 ICER、中断挂起寄存器 ISPR、中断挂起清除寄存器 ICPR、中断激活寄存器 IABR 和中断优先级寄存器 IPR。其中 ISER 和 ICER 用于中断屏蔽管理，即是否允许中断请求；ISPR 和 ICPR 用于中断挂起管理，设置挂起或清除挂起状态；这两组寄存器各有 8 个，STM32L152 只支持 81 个外部中断，因此中断屏蔽和中断挂起只有 3 对寄存器，分别为 ISER0～ISER2、ICER0～ICER2、ISPR0～ISPR2、ICPR0～ICPR2。IABR 有 3 个寄存器 IABR0～IABR2，IPR 有 81 个 8bit 的寄存器 IPR0～IPR80。所有的 NVIC 寄存器都可采用字节、半字和字方式进行访问，不管处理器存储字节的顺序如何，所有 NVIC 寄存器和系统调试寄存器都是采用小端(little endian)字节排列顺序，即低位字节存储在低地址。另外，下列寄存器也对中断处理有重大影响：

- 全局异常屏蔽寄存器(PRIMASK，FAULTMASK 以及 BASEPRI)；
- 向量表偏移量寄存器；
- 优先级分组位段；
- 软件触发中断寄存器。

1) 中断使能寄存器 (NVIC_ISERx)，x=0,1,2

NVIC 支持可屏蔽中断，即可以通过控制中断使能寄存器和中断清除寄存器对每个外部中断源进行独立的控制。中断使能寄存器的有效域定义如图 6-8 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SETENA[31:16]															
rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SETENA[15:0]															
rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs

图 6-8　中断使能寄存器

中断使能位 SETENA[31：0]，写 1 使能中断，写 0 无效。读该寄存器时，对应位为 1 表示该中断已使能，0 表示该中断没有使能。

2）中断清除寄存器（NVIC_ICERx），x=0，1，2

中断清除寄存器的有效域定义如图 6-9 所示。中断清除位 CLRENA[31：0]，写 1 表示禁止该中断，写 0 无效；读取该寄存器，对应位为 1 表明相应的中断被使能，否则被禁用。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CLRENA[31:16]															
rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CLRENA[15:0]															
rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1

图 6-9　中断清除寄存器

3）中断挂起设置寄存器（NVIC_ISPRx），x=0，1，2

一个发生的中断如果不能被立即响应，就称它被挂起，被挂起的中断由挂起状态寄存器保持，保证中断源释放了中断请求信号后中断请求也不会丢失。

中断挂起寄存器的有效域定义如图 6-10 所示。中断挂起设置位 SETPEND[31：0]，写 1 则相应中断被挂起，写 0 无效，读取该寄存器的值，对应位为 1 表明该中断正在挂起中。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SETPEND[31:16]															
rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SETPEND[15:0]															
rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs

图 6-10　中断挂起设置寄存器

当一个中断正在挂起时，写 NVIC_ISPRx 寄存器的相应位为 1 对该中断没有影响；当写入 NVIC_ISPRx 寄存器相应位为 1 但该中断被禁用时，将相应中断置为挂起状态。

4）中断挂起清除寄存器（NVIC_ICPRx），x=0，1，2

中断挂起清除寄存器的有效域定义如图 6-11 所示。中断挂起清除位 CLRPEND[31：0]，写 1 时清楚挂起的中断，写 0 无效；读取该寄存器，相应位为 1 时表明中断正在挂起中。

5）中断活动寄存器（NVIC_IABRx），x=0，1，2

中断活动寄存器的有效域定义如图 6-12 所示。中断激活状态 ACTIVE[31：0]，1 表示中断处于激活状态或激活挂起状态，0 表示中断没有被激活。如果一个挂起的中断被使能，

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CLRPEND[31:16]															
rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CLRPEND[15:0]															
rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1

图 6-11 中断挂起设置寄存器

NVIC 将激活该中断。每个外部中断都有一个活动状态位。在处理器执行了其 ISR 的第一条指令后，它的活动位就被置 1，并且直到 ISR 返回时才硬件清零。由于支持嵌套，允许高优先级异常抢占某个 ISR。然而，哪怕一个中断被抢占，其活动状态也依然为 1。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
ACTIVE[31:16]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ACTIVE[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

图 6-12 中断活动寄存器

6）中断优先级寄存器(NVIC_IPRx)，x=0，1，…，80

每一个中断优先级寄存器 IPRx 包括 1 个中断的优先级，优先级的设置参见表 6-2 中断优先级分组，每个 IPRx 寄存器是一个字节，如图 6-13 所示。STM32 只使用 8bit 中的 4 个比特，即 IPRx 的低 4 位为 0。

	31　　24	23　　16	15　　8	7　　0
E000E400	PRI_3	PRI_2	PRI_1	PRI_0
E000E404	PRI_7	PRI_6	PRI_5	PRI_4
E000E408	PRI_11	PRI_10	PRI_9	PRI_8
E000E40C	PRI_15	PRI_14	PRI_13	PRI_12
E000E410	PRI_19	PRI_18	PRI_17	PRI_16
E000E414	PRI_23	PRI_22	PRI_21	PRI_20
E000E418	PRI_27	PRI_26	PRI_25	PRI_24
E000E41C	PRI_31	PRI_30	PRI_29	PRI_28

图 6-13 中断优先级寄存器

7）软件中断触发寄存器(NVIC_STIR)

软件触发中断寄存器的有效域定义如图 6-14 所示。中断号 NTID[8：0]，用于指示软

件产生的中断编号。INTID[8：0]中写入 0～239 的数值，产生一个对应于该编号的中断。

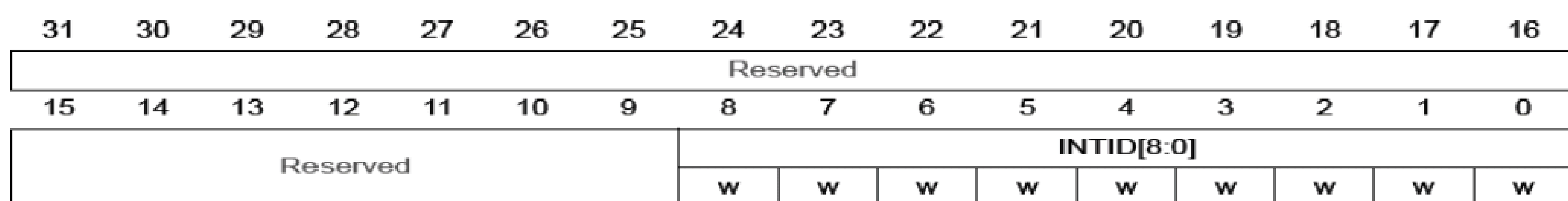


图 6-14　软件触发中断寄存器

【思考题：为何中断使能、清除，中断挂起、挂起清除要使用成对的寄存器控制？】

当有中断发生时，外围设备将发送中断信号给 NVIC，如果 NVIC 收到中断信号且该中断被使能，但中断没有被激活，将该中断置到挂起状态；当处理器响应该中断并执行中断服务程序时，挂起状态的中断被激活，ISR 执行完成后，中断状态被置为没有激活状态；若在 ISR 执行过程中，中断信号再次到达，ISR 执行完成后，将该中断置为挂起状态，等待再次响应该中断。

6.4.3　系统异常处理

系统异常指的是异常向量表前 15 个异常，NVIC 中，系统异常的处理不采用上述的 5 类寄存器，而是由中断控制状态寄存器 ICSR、应用中断/复位控制寄存器 AIRCR、系统控制寄存器 SCR、配置控制寄存器 CCR、系统异常 4～7 优先级寄存器 SHPR1、系统异常 8～11 优先级寄存器 SHPR2、系统异常 12～15 优先级寄存器 SHPR3、系统异常控制与状态寄存器 SHCSR 等对中断使能、清除、挂起设置、挂起清除、优先级、激活状态等进行控制和描述。

其中，应用中断与复位控制寄存器 AIRCR 用于决定数据的字节顺序、清除所有有效的状态信息、执行系统复位、改变优先级分组位置等功能，其寄存器定义如图 6-15 所示。

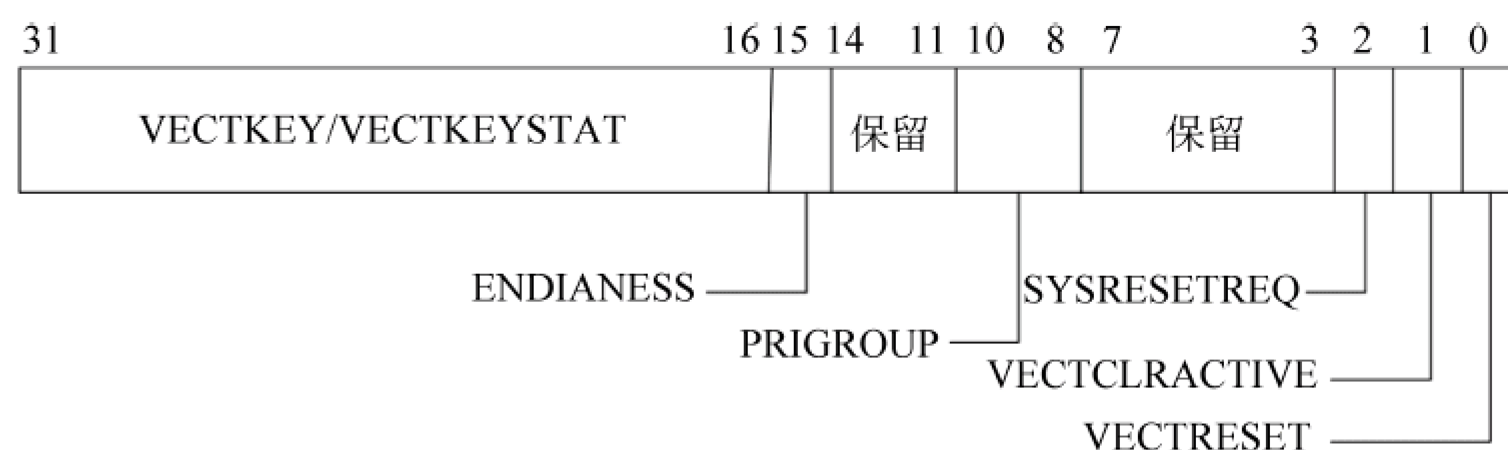


图 6-15　应用中断与复位控制寄存器

PRIGROUP 用于配置中断优先级分组，其取值为：

0　7.1 表示 7 位抢占式优先级，1 位子优先级；

1　6.2 表示 6 位抢占式优先级，2 位子优先级；

2　5.3 表示 5 位抢占式优先级，3 位子优先级；

3　4.4 表示 4 位抢占式优先级，4 位子优先级；

4　3.5 表示 3 位抢占式优先级，5 位子优先级；

5　2.6 表示 2 位抢占式优先级，6 位子优先级；

6　1.7 表示 1 位抢占式优先级，7 位子优先级；

7　0.8 表示 0 位抢占式优先级，8 位子优先级。

用法 Fault，总线 Fault 以及存储器管理 Fault 都是特殊的异常，它们的使能、挂起和激活控制是通过系统异常控制及状态寄存器(SHCSR)实现的。此外，SHCSR 寄存器还用于管理 SVC、SysTick、SendSV、监控异常的激活状态。NMI、SysTick 定时器以及 PendSV 的挂起通过 ICSR 寄存器配置，NMI 和硬件错误无需使能就可以发生。

系统异常的优先级由 SHPR1～SHPR3 寄存器设置，可设置的优先级为 0～31。SHPR1～SHPR3 可按字节访问，如表 6-8 所示。

表 6-8　系统异常优先级寄存器

地　址	名　称	说　明
0xE000_ED18	PRI_4	存储器管理 Fault 的优先级
0xE000_ED19	PRI_5	总线 Fault 的优先级
0xE000_ED1A	PRI_6	用法 Fault 的优先级
0xE000_ED1F	PRI_11	SVC 优先级
0xE000_ED20	PRI_12	调试监控优先级
0xE000_ED22	PRI_14	PendSV 的优先级
0xE000_ED23	PRI_15	SysTick 的优先级

6.4.4　全局中断管理

Cortex-M3 的异常/中断屏蔽寄存器组有 3 个寄存器用于全局中断管理。

- PRIMASK 寄存器只有最低位有效，该寄存器置为 1 后，就关掉所有可屏蔽异常，只剩下 NMI 和硬件 Fault 可以响应。其默认值是 0，表示没有关闭中断。
- FAULTMASK 寄存器为单一比特的寄存器。置为 1 后，只有 NMI 可以响应。默认值为 0，表示没有关异常。
- BASEPRI 寄存器低 9 位有效，定义了被屏蔽优先级的阈值。当它被设置为某个值后，所有优先级编号大于或等于此值的中断(即优先级化中断)都被关。若设置成 0，则不关断任何中断，0 为默认值。

我们可以使用 MRS/MSR 指令访问这三个寄存器：

```
MRS     R0, BASEPRI     ;读取 BASEPRI 到 R0 中
MOV     R1, #20
MSR     BASEPRI,R1      ;将 20 数据写入到 BASEPRI 中
```

Cortex-M3 还专门设置了 CPS 指令用于 PRIMASK 和 FAULTMASK 的操作：

```
CPSID    I     ;PRIMASK=1,关中断
```

```
CPSIE    I    ;PRIMASK=0,开中断
CPSID    F    ;FAULTMASK=1,关异常
CPSIE    F    ;FAULTMASK=0,开异常
```

在 CMSIS 库中,core_cmFunc.h 中提供了以下函数用于上述寄存器的操作:

- void __set_BASEPRI(uint32_t basePri);
- uint32_t __get_BASEPRI(void);
- void __set_PRIMASK(uint32_t priMask);
- uint32_t __get_PRIMASK(void);
- void __set_FAULTMASK(uint32_t faultMask);
- uint32_t __get_FAULTMASK(void);

针对 C 语言,我们常用的两个全局中断开关函数为:

- void __disable_irq(void);
- void __enable_irq(void);

中断向量表默认存储在 0 地址,0 地址是 ROM 区,CM3 允许向量表重定位,中断向量表可以放置到 RAM 区,这样可以方便在程序中对中断向量表进行修改。NVIC 控制器的向量表偏移量寄存器 VTOR(地址 0xE000ED08)用于配置中断向量表的存储地址。VTOR 的定义如表 6-9 所示。

表 6-9　偏移量寄存器的定义

位段	名称	类型	复位值	描　　述
29	TBLBASE	RW	0	向量表在 Code 区(0),还是在 RAM 区(1)
15	ENDIANESS	R	—	向量表的起始地址,一般置为 0

6.4.5　NVIC 库函数

NVIC 库函数提供使能或者失能 IRQ 中断,使能或者失能单独的 IRQ 通道,改变 IRQ 通道的优先级等功能。

CMSIS 库中,NVIC 寄存器结构 NVIC_TypeDeff 在文件 core_cm3.h 中的定义如下:

```
typedef struct
{
    __IO uint32_t ISER[8];
    uint32_t RESERVED0[24];
    __IO uint32_t ICER[8];
    uint32_t RSERVED1[24];
    __IO uint32_t ISPR[8];
    uint32_t RESERVED2[24];
    __IO uint32_t ICPR[8];
```

```
    uint32_t RESERVED3[24];
    __IO uint32_t IABR[8];
    uint32_t RESERVED4[56];
    __IO uint8_t  IP[240];
    uint32_t RESERVED5[644];
    __O  uint32_t STIR;
} NVIC_Type;
```

操作 NVIC 的指针定义如下：

```
#define SCS_BASE    (0xE000E000UL)              /* System Control Space Base Address */
#define NVIC_BASE   (SCS_BASE +  0x0100UL)      /* NVIC Base Address */
#define NVIC        ((NVIC_Type *)NVIC_BASE)    /* NVIC configuration struct */
```

嵌套向量中断控制器 NVIC 的相关库函数定义在 misc. c 和 core_cm3. h 中，如表 6-10 所示。

表 6-10　NVIC 操作库函数

函　数　名	描　　述
NVIC_PriorityGroupConfig	设置优先级分组，抢占优先级和从优先级的位数
NVIC_Init	根据 NVIC_InitStruct 中指定的参数初始化外设 NVIC 寄存器
NVIC_SetVectorTable	设置向量表的位置和偏移
NVIC_SetPriorityGrouping	core_cm3 提供的设置中断优先级分组函数
NVIC_GetPriorityGrouping	core_cm3 提供的获取中断优先级分组函数
NVIC_EnableIRQ	使能一个外部中断
NVIC_DisableIRQ	清除一个外部中断
NVIC_GetPendingIRQ	获取某一外部中断的挂起状态
NVIC_SetPendingIRQ	设置某一外部中断的挂起状态
NVIC_ClearPendingIRQ	清除某一外部中断的挂起状态
NVIC_GetActive	获取某一外部中断的激活状态
NVIC_SetPriority	设置中断优先级(包括外部中断和系统异常)
NVIC_GetPriority	获取中断优先级(包括外部中断和系统异常)
NVIC_EncodePriority	根据优先级分组将抢占优先级和从优先级生成优先级
NVIC_DecodePriority	根据优先级分组将优先级分解为抢占优先级和从优先级

各个函数的具体功能如下：

1) void NVIC_PriorityGroupConfig(uint32_t NVIC_PriorityGroup)

该函数用于设置嵌套向量中断控制器的优先级分组。参数 NVIC_PriorityGroup 的取值为 0～4，分别对应中断优先级分组表中的各种分组情况。

```
#define NVIC_PriorityGroup_0    ((uint32_t)0x700)    //0 比特抢占优先级,4bit 从优先级
#define NVIC_PriorityGroup_1    ((uint32_t)0x600)    //1 比特抢占优先级,3bit 从优先级
#define NVIC_PriorityGroup_2    ((uint32_t)0x500)    //2 比特抢占优先级,2bit 从优先级
#define NVIC_PriorityGroup_3    ((uint32_t)0x400)    //3 比特抢占优先级,1bit 从优先级
#define NVIC_PriorityGroup_4    ((uint32_t)0x300)    //4 比特抢占优先级,0bit 从优先级
```

如果不执行该函数,默认的分组方式是分组 0,抢占优先级为 0,可以设置从优先级为 0～15 共 16 种情况。

2) void NVIC_Init(NVIC_InitTypeDef * NVIC_InitStruct)

该函数用于对嵌套向量中断控制器进行初始化。所有初始化信息都通过结构体指针 NVIC_InitStruct 进行传递,该指针指向 NVIC_InitTypeDef 类型。

NVIC_InitTypeDef 定义如下:

```
typedef struct
{
    uint8_t NVIC_IRQChannel;
    uint8_t NVIC_IRQChannelPreemptionPriority;
    uint8_t NVIC_IRQChannelSubPriority;
    FunctionalState NVIC_IRQChannelCmd;
} NVIC_InitTypeDef;
```

其中,NVIC_IRQChannel 设置中断通道,其取值由 stm32L1xx.h 中的 IRQn 枚举类型定义。

```
typedef enum IRQn
{
    NonMaskableInt_IRQn          = -14,
    MemoryManagement_IRQn        = -12,
    BusFault_IRQn                = -11,
    UsageFault_IRQn              = -10,
    SVC_IRQn                     = -5,
    DebugMonitor_IRQn            = -4,
    PendSV_IRQn                  = -2,
    SysTick_IRQn                 = -1,
    WWDG_IRQn                    = 0,
    PVD_IRQn                     = 1,
    TAMPER_STAMP_IRQn            = 2,
    RTC_WKUP_IRQn                = 3,
    RCC_IRQn                     = 5,
    EXTI0_IRQn                   = 6,
    EXTI1_IRQn                   = 7,
    EXTI2_IRQn                   = 8,
    EXTI3_IRQn                   = 9,
    EXTI4_IRQn                   = 10,
    DMA1_Channel1_IRQn           = 11,
```

```
        DMA1_Channel2_IRQn            = 12,
        DMA1_Channel3_IRQn            = 13,
        DMA1_Channel4_IRQn            = 14,
        ...
    } IRQn_Type;
```

NVIC_IRQChannelPreemptionPriority 设置抢占优先级，NVIC_IRQChannelSubPriority 设置从优先级，其取值范围均为 0～15；NVIC_IRQChannelCmd 设置 NVIC_IRQChannel 指定的中断通道使能，其取值为 ENABLE 和 DISABLE。

3）void NVIC_SetVectorTable(uint32_t NVIC_VectTab, uint32_t Offset)

该函数用于设置中断向量表的位置和偏移量，参数 NVIC_VectTab 用于指定中断向量表在 RAM 中还是 Flash 中，其取值为：NVIC_VectTab_RAM 和 NVIC_VectTab_FLASH；参数 Offset 用于指定中断向量表的偏移量，一般设定为 0。

4）void NVIC_SetPriorityGrouping(uint32_t PriorityGroup)

该函数由 core_cm3.h 定义，用于配置优先级分组，参数 ProrityGroup 的取值范围为 0～7，分别表示图 6-3 中 PRIGROUP 的八种抢占优先级和从优先级的分组情况。在 STM32L152 中，由于只使用了 4 个 bit 作为优先级配置，因此我们使用 misc.c 中提供 NVIC_PriorityGroupConfig 函数对优先级分组进行配置。

5）uint32_t NVIC_GetPriorityGrouping(void)

该函数用于读取 NVIC 控制器的优先级分组配置，其返回值为 0～7，含义见图 6-3 的 PRIGROUP 定义。

6）void NVIC_EnableIRQ(IRQn_Type IRQn)

该函数用于使能外部中断，输入参数 IRQn 为 0～239，即只能使能系统异常以外的外部中断，IRQn 的具体定义见 NVIC_Init 函数的 IRQn_Type 枚举类型定义。

7）void NVIC_DisableIRQ(IRQn_Type IRQn)

该函数用于屏蔽外部 IRQ 中断，输入参数 IRQn 为 0～239。

8）uint32_t NVIC_GetPendingIRQ(IRQn_Type IRQn)

该函数用于读取中断挂起寄存器并返回对应的 IRQn 编号的外部中断是否被挂起，若挂起，返回值为 1，否则返回值为 0. IRQn 必须为 0～239 的编号值。

9）void NVIC_SetPendingIRQ(IRQn_Type IRQn)

该函数用于配置挂起寄存器使得一个外部中断处于挂起状态，输入参数 IRQn 为要挂起的外部中断中断号，IRQn 的取值范围为 0～239。

10）NVIC_ClearPendingIRQ(IRQn_Type IRQn)

该函数用于清除一个外部中断的挂起位，输入参数 IRQn 的取值范围为 0～239。

11）uint32_t NVIC_GetActive(IRQn_Type IRQn)

该函数用于读取中断激活寄存器，返回输入参数 IRQn 的激活位，1 表示该外部中断处于激活状态，0 表示未激活，IRQn 的取值范围为 0～239。

12）void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)

该函数用于设置中断优先级，输入参数 IRQn 为中断号，取值范围为－14～239，即该函

数可以配置所有异常和中断的优先级，参数 priority 为所要设置的中断优先级，注意系统异常和外部中断的优先级值取值范围。

13）uint32_t NVIC_GetPriority(IRQn_Type IRQn)

该函数用于读取某个异常或中断的优先级，输入参数 IRQn 的取值范围为－14～239。

14）uint32_t NVIC_EncodePriority (uint32_t PriorityGroup, uint32_t PreemptPriority, uint32_t SubPriority)

该函数用于根据 PriorityGroup 参数指定的优先级分组，将抢占优先级 PreemptPriority 和从优先级 SubPriority 组合成一个 8bit 的优先级值。每种处理器实现 cortexM3 的优先级时可以不使用所有的 8bit(256 种优先级)，其使用多少种优先级由__NVIC_PRIO_BITS 决定，在 STM32L152 中__NVIC_PRIO_BITS ＝4，即只支持 16 种优先级。如果该函数中的 PriorityGroup 值和之前设定的值不同，则以这两个值中的较小的值为准。

15）void NVIC_DecodePriority (uint32_t Priority, uint32_t PriorityGroup, uint32_t * pPreemptPriority, uint32_t * pSubPriority)

该函数用于根据输入的优先级 Priority 和优先级分组 PriorityGroup，将 Priority 分解成抢占优先级和从优先级。

【例 6-1】　中断控制器初始化案例。

```
void NVIC_Init(NVIC_InitTypeDef* NVIC_InitStruct)
{
  uint8_t tmppriority = 0x00, tmppre = 0x00, tmpsub = 0x0F;
  if (NVIC_InitStruct->NVIC_IRQChannelCmd != DISABLE)
  {  //计算优先级
    tmppriority = (0x700 - ((SCB->AIRCR) & (uint32_t)0x700))>>0x08;
    tmppre = (0x4 - tmppriority);
    tmpsub = tmpsub >> tmppriority;
    tmppriority = (uint32_t)NVIC_InitStruct->NVIC_IRQChannelPreemptionPriority << tmppre;
    tmppriority |= (uint8_t) (NVIC_InitStruct->NVIC_IRQChannelSubPriority & tmpsub);
    tmppriority = tmppriority << 0x04;
    NVIC->IP[NVIC_InitStruct->NVIC_IRQChannel] = tmppriority;
     //使能中断
    NVIC->ISER[NVIC_InitStruct->NVIC_IRQChannel >> 0x05] =
      (uint32_t)0x01 << (NVIC_InitStruct->NVIC_IRQChannel & (uint8_t)0x1F);
  }
  else
  { //清除中断使能位
    NVIC->ICER[NVIC_InitStruct->NVIC_IRQChannel >> 0x05] =
      (uint32_t)0x01 << (NVIC_InitStruct->NVIC_IRQChannel & (uint8_t)0x1F);
  }
}
```

【例 6-2】　采用库函数的 NVIC 中断配置实例如下所示。

```
void NVIC_Configuration(void)
{
  NVIC_InitTypeDef NVIC_InitStructure;
#ifdef  VECT_TAB_RAM
  /* 中断向量表位于 0x20000000 */
  NVIC_SetVectorTable(NVIC_VectTab_RAM, 0x0);
#else  /* VECT_TAB_FLASH  */
  /* 中断向量表位于 0x08000000 */
  NVIC_SetVectorTable(NVIC_VectTab_FLASH, 0x0);
#endif
  /* 中断优先级分组配置 */
  NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
  /* 使能 EXTI9_5 中断 */
  NVIC_InitStructure.NVIC_IRQChannel = EXTI9_5_IRQn
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);
}
```

【例 6-3】 采用库函数配置多个中断的实例如下所示。

```
/* 中断优先级分组配置 */
  NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);
  /* 配置 TIM2 中断 */
  NVIC_InitStructure.NVIC_IRQChannel = TIM2_IRQn;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);
  /* 配置 TIM3 中断 */
  NVIC_InitStructure.NVIC_IRQChannel = TIM3_IRQn;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
  NVIC_Init(&NVIC_InitStructure);
  /* 配置 TIM4 中断 */
  NVIC_InitStructure.NVIC_IRQChannel = TIM4_IRQn;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 2;
  NVIC_Init(&NVIC_InitStructure);
```

6.5 外部中断/事件控制器 EXTI

外部中断是由处理器的 I/O 引脚产生的中断，由于处理器的 I/O 引脚较多，使用也比

较灵活,为了便于管理这些 I/O 的中断,STM32L152 提供了一个外部中断/事件控制器专门处理外部中断。

外部中断/事件控制器由 23 个产生中断/事件请求的边沿检测器组成,其中 16 用于外部 I/O 引脚中断,7 个用于特殊事件处理,每个输入线可以独立地配置对应的触发事件,即上升沿、下降沿或双边沿触发。每个输入线都可以独立地被屏蔽。挂起位保持着中断请求的状态。外部中断/事件控制器的框图,如图 6-16 所示。

【思考题:中断和事件有何区别?】

由图 6-16 可以看出:

(1) 如果要产生中断,必须对中断线进行配置并使能该中断线。根据需要的边沿检测设置 2 个触发寄存器,同时在中断屏蔽寄存器相应位写 1 来允许中断请求。当外部中断线上发生了期待的边沿时,将产生一个中断请求,对应的挂起位被置为 1。

(2) 如果要产生事件,必须对事件线进行配置并使能该事件线。根据需要的边沿检测设置 2 个触发寄存器,同时在事件屏蔽寄存器相应位写 1 来允许事件请求。当事件线上发生了需要的边沿时,将产生一个事件请求脉冲,对应的挂起位不被置 1。

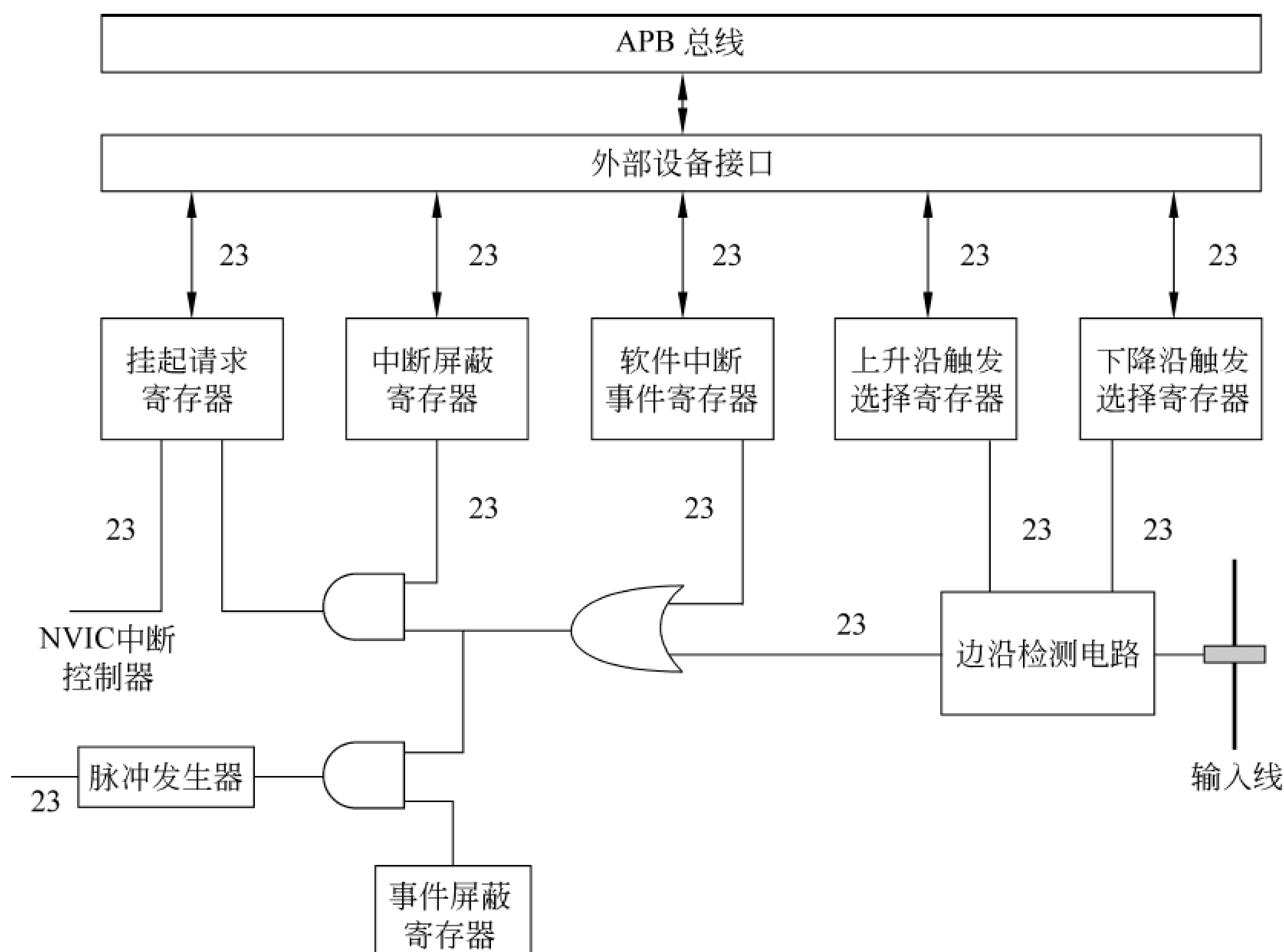


图 6-16　外部中断/事件控制器框图

(3) 通过软件向软件中断/事件寄存器写 1,也可以产生中断/事件请求。

EXTI 相关寄存器包括中断屏蔽寄存器(EXTI_IMR)、事件屏蔽寄存器(EXTI_EMR)、上升沿触发选择寄存器(EXTI_RTSR)、下降沿触发选择寄存器(EXTI_FTSR)、软件中断事件寄存器(EXTI_SWIER)、挂起寄存器(EXTI_PR)。

硬件中断配置过程:

(1) 配置中断线的屏蔽位(EXTI_IMR);

(2) 配置所选中断线的触发选择位(EXTI_RTSR 和 EXTI_FTSR);

(3) 配置对应到外部中断控制器(EXTI)的 NVIC 中断通道的使能和屏蔽位,使得中断线的请求可以被正确地响应。

硬件事件配置过程:

(1) 配置事件线的屏蔽位(EXTI_EMR);

(2) 配置所选事件线的触发选择位(EXTI_RTSR 和 EXTI_FTSR)。

软件中断/事件配置过程:

(1) 配置中断/事件线的屏蔽位(EXTI_IMR 和 EXTI_EMR);

(2) 设置软件中断寄存器的请求位(EXTI_SWIER)。

6.6 寄存器说明

外部中断控制器的寄存器如表 6-11 所示。

表 6-11 EXTI 寄存器说明

寄存器名称	偏移量	功能	复位值
中断屏蔽寄存器 EXTI_IMR	0x00	设定中断是否屏蔽	0x 0000 0000
事件屏蔽寄存器 EXTI_EMR	0x04	设定事件是否屏蔽	0x 0000 0000
上升沿触发选择寄存器 EXTI_RTSR	0x08	配置中断/事件上升沿触发	0x 0000 0000
下降沿触发选择寄存器 EXTI_FTSR	0x0C	配置中断/事件下降沿触发	0x 0000 0000
软件触发中断寄存器 EXTI_SWIER	0x10	控制软件触发中断/事件	0x 0000 0000
挂起寄存器 EXTI_PR	0x14	中断的挂起状态	0x xxxx xxxx

1) 中断屏蔽寄存器(EXTI_IMR)

中断屏蔽寄存器是 32 位的寄存器,偏移地址为 0x00,复位值为 0x00000000,其有效位定义如图 6-17 所示。位 24~31 保留,必须始终保持为复位状态(0)。位 0~23 中位 x 的值为 0,则屏蔽来自线 x 的中断请求,位 0~23 中位 x 的值为 1,则开放来自线 x 的中断请求。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								MR23	MR22	MR21	MR20	MR19	MR18	MR17	MR16
								rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MR15	MR14	MR13	MR12	MR11	MR10	MR9	MR8	MR7	MR6	MR5	MR4	MR3	MR2	MR1	MR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 6-17 中断屏蔽寄存器

2) 事件屏蔽寄存器(EXTI_EMR)

事件屏蔽寄存器是 32 位的寄存器,偏移地址为 0x04,复位值为 0x00000000,其有效位

定义如图 6-18 所示。位 24～31 保留，必须始终保持为复位状态(0)。位 0～23 中位 x 的值为 0，则屏蔽来自线 x 的事件请求，位 0～23 中位 x 的值为 1，则开放来自线 x 的事件请求。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								MR23	MR22	MR21	MR20	MR19	MR18	MR17	MR16
								rw	rw	rw	rw	rw	rw	rw	rw
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
MR15	MR14	MR13	MR12	MR11	MR10	MR9	MR8	MR7	MR6	MR5	MR4	MR3	MR2	MR1	MR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 6-18　事件屏蔽寄存器

3）上升沿触发选择寄存器(EXTI_RTSR)

上升沿触发选择寄存器是 32 位的寄存器，偏移地址为 0x08，复位值为 0x00000000，其有效位定义如图 6-19 所示。位 24～31 保留，必须始终保持为复位状态(0)。位 0～23 中位 x 的值为 0，则禁止输入线 x 上的上升沿触发中断或事件，位 0～23 中位 x 的值为 1，则允许输入线 x 上的上升沿触发中断或事件。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								TR23	TR22	TR21	TR20	TR19	TR18	TR17	TR16
								rw	rw	rw	rw	rw	rw	rw	rw
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
TR15	TR14	TR13	TR12	TR11	TR10	TR9	TR8	TR7	TR6	TR5	TR4	TR3	TR2	TR1	TR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 6-19　上升沿触发选择寄存器

4）下降沿触发选择寄存器(EXTI_FTSR)

下降触发选择寄存器是 32 位的寄存器，偏移地址为 0x0C，复位值为 0x00000000，其有效位定义如图 6-20 所示。位 24～31 保留，必须始终保持为复位状态(0)。位 0～23 中位 x 的值为 0，则禁止输入线 x 上下降沿触发中断或事件，位 0～23 中位 x 的值为 1，则允许输入线 x 上的下降沿触发中断或事件。

5）软件中断事件寄存器(EXTI_SWIER)

软件中断事件寄存器是 32 位的寄存器，偏移地址为 0x10，复位值为 0x00000000，其有效位定义如图 6-21 所示。位 24～31 保留，必须始终保持为复位状态(0)。位 0～23 中位 x 的值为 0，且相应的 EXTI_IMR 或 EXTI_EMR 寄存器中的中断位是 1(即开放来自线 x 的

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								TR23	TR22	TR21	TR20	TR19	TR18	TR17	TR16
								rw	rw	rw	rw	rw	rw	rw	rw
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
TR15	TR14	TR13	TR12	TR11	TR10	TR9	TR8	TR7	TR6	TR5	TR4	TR3	TR2	TR1	TR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 6-20　下降沿触发选择寄存器

中断或事件)，则写入 1 产生一个中断或事件请求。该位通过设置挂起寄存器 EXTI_PR 中的相应位 1 来清零。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								SWIER 23	SWIER 22	SWIER 21	SWIER 20	SWIER 19	SWIER 18	SWIER 17	SWIER 16
								rw	rw	rw	rw	rw	rw	rw	rw
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
SWIER 15	SWIER 14	SWIER 13	SWIER 12	SWIER 11	SWIER 10	SWIER 9	SWIER 8	SWIER 7	SWIER 6	SWIER 5	SWIER 4	SWIER 3	SWIER 2	SWIER 1	SWIER 0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 6-21 软件触发中断寄存器

6）挂起寄存器(EXTI_PR)

软件中断事件寄存器是 32 位的寄存器，偏移地址为 0x14，复位值为 0x00000000，其有效位定义如图 6-22 所示。位 24～31 保留，必须始终保持为复位状态(0)。位 0～23 中位 x 的值为 0，则表明没有相应线的中断或事件请求产生。位 0～23 中位 x 的值为 1，则表明相应线的中断或事件请求已经产生。当向相应位写入 1，可以进行清零。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								PR23	PR22	PR21	PR20	PR19	PR18	PR17	PR16
								rw	rw	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
PR15	PR14	PR13	PR12	PR11	PR10	PR9	PR8	PR7	PR6	PR5	PR4	PR3	PR2	PR1	PR0
rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1

图 6-22 挂起状态寄存器

由于 GPIO 引脚多，EXTI I/O 中断引脚少，因此需要对中断线和 I/O 进行映射，其映射关系如图 6-23 所示。

由图 6-23 可以看出，线 EXTIx(x=0，…，15)通过多路选择器来选择一个 GPIO 口 x 位作为中断。同一个时刻，不同端口的同一序号只能设置其中一个作为中断。16 个外部 I/O 中断均可以映射到不同的 I/O 引脚上，寄存器 SYSCFG_EXTICRn(n=1，2，3，4)用于 I/O 引脚的中断映射配置，每个 I/O 引脚占用 4 位，如 EXTI0 中断可配置到 PA0，PB0，PC0，…，PG0 八个 I/O 中的一个，寄存器有效位定义如图 6-24 所示。

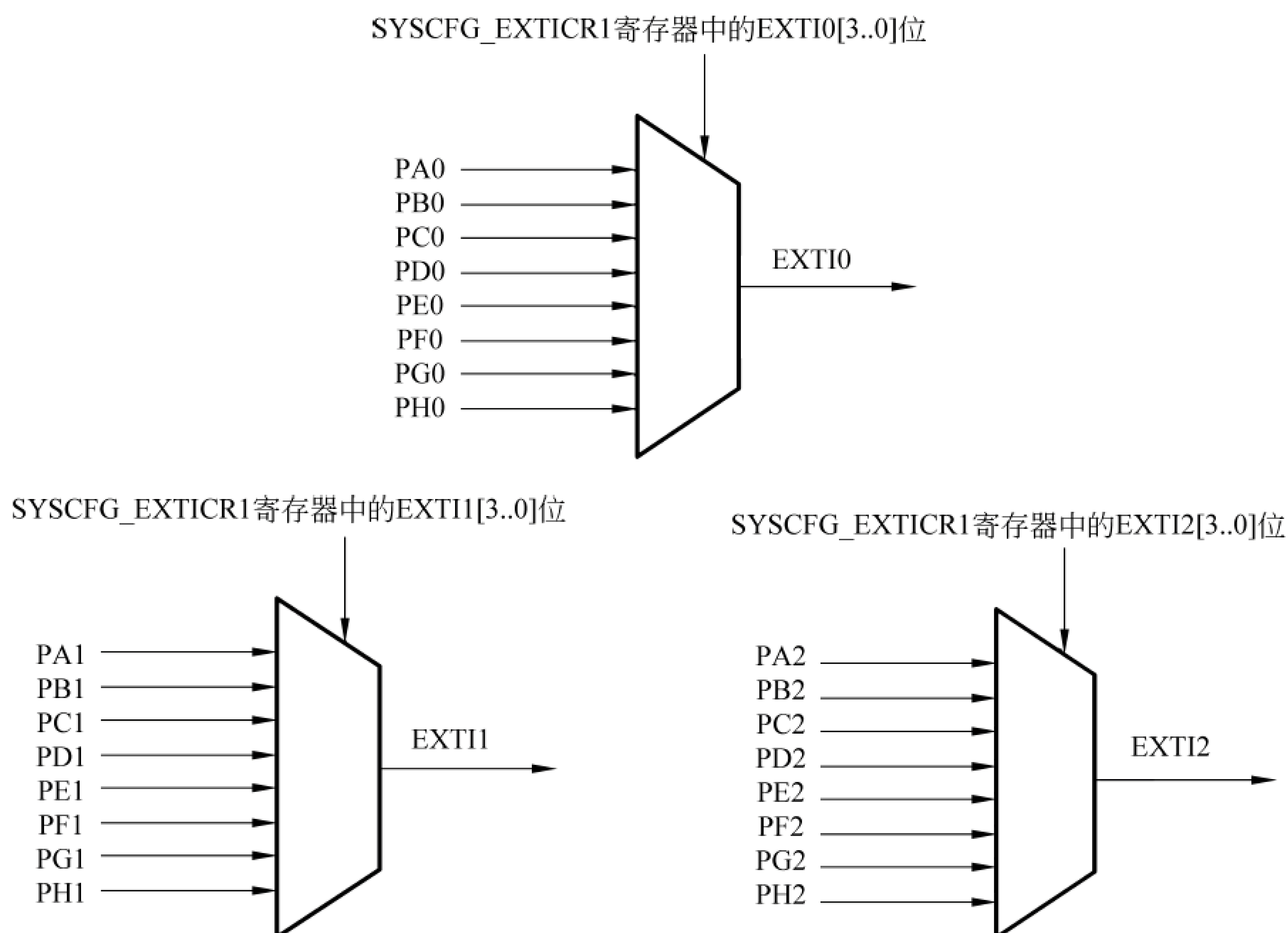


图 6-23　外部中断通用输入输出口映射

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EXTI3[3:0]				EXTI2[3:0]				EXTI1[3:0]				EXTI0[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 6-24　外部中断源选择寄存器

SYSCFG_EXTICR2 用于配置 EXTI4～EXTI7，SYSCFG_EXTICR3 用于配置 EXTI8～EXTI11，SYSCFG_EXTICR4 用于配置 EXTI2～EXTI5。

其他的中断线分配如下：

(1) EXTI 中断线 16 连接到 PVD 输出；

(2) EXTI 中断线 17 连接到 RTC 闹钟事件；

(3) EXTI 中断线 18 连接到 USB 设备唤醒事件；

(4) EXTI 中断线 19 连接到 RTC Tamper 和 TimeStamp 事件；

(5) EXTI 中断线 20 连接到 RTC 唤醒事件；

(6) EXTI 中断线 21 连接到 Comparator1 唤醒事件；

(7) EXTI 中断线 22 连接到 Comparator2 唤醒事件；

(8) EXTI 中断线 23 连接到通道获取中断。

6.7 EXTI 函数库

EXTI 寄存器结构 EXTI_TypeDef 在文件 stm32L1xx.h 中定义。

```
typedef struct
{
    __IO uint32_t IMR;        /* 外部中断屏蔽寄存器 */
    __IO uint32_t EMR;        /* 外部事件屏蔽寄存器 */
    __IO uint32_t RTSR;       /* 外部中断/事件上升沿配置寄存器 */
    __IO uint32_t FTSR;       /* 外部中断/事件下降沿配置寄存器 */
    __IO uint32_t SWIER;      /* 软件中断/事件配置寄存器 */
    __IO uint32_t PR;         /* 中断/事件挂起寄存器 */
} EXTI_TypeDef;
```

外部中断的库函数如表 6-12 所示。

表 6-12　外部中断 EXTI 的相关的库函数

函　数　名	功　　能
EXTI_DeInit	将外设 EXTI 寄存器设为默认值
EXTI_Init	用 EXTI_InitStruct 中指定的参数初始化外设 EXTI 寄存器
EXTI_StructInit	将 EXTI_InitStruct 中的每一个参数按默认值填入
EXTI_GenerateSWInterrupt	产生一个软中断
EXTI_GetFlagStatus	检查指定的 EXTI 挂起寄存器
EXTI_ClearFlag	清除 EXTI 挂起寄存器
EXTI_GetITStatus	检查指定的 EXTI 线路触发请求发生与否
EXTI_ClearITPendingBit	清除 EXTI 线路挂起位

1. 函数 EXTI_Init

功能描述：根据 EXTI_InitStruct 中指定的参数初始化外设 EXTI 寄存器。

函数原型：void EXTI_Init(EXTI_InitTypeDef * EXTI_InitStruct)。

输入参数 EXTI_InitStruct：指向结构 EXTI_InitTypeDef 的指针，包含了外设 EXTI 的配置信息，EXTI_InitTypeDef 的定义如下：

```
typedef struct
{
    uint32_t EXTI_Line;
    EXTIMode_TypeDef    EXTI_Mode;
    EXTITrigger_TypeDef    EXTI_Trigger;
```

```
    FunctionalState    EXTI_LineCmd;
} EXTI_InitTypeDef;
```

EXTI_Line 选择待配置的外部中断中断线，其取值为 EXTI_Linex，其中 x 为 0～15。

EXTI_Mode 设置待配置中断线的中断模式，其取值为：

- EXTI_Mode_Event：设置 EXTI 中断线为事件请求；
- EXTI_Mode_Interrupt：设置 EXTI 中断线为中断请求。

EXTI_Trigger 设置待配置中断线的边沿触发模式，其取值为：

- EXTI_Trigger_Falling：设置输入中断线为下降沿触发；
- EXTI_Trigger_Rising：设置输入中断线为上升沿触发；
- EXTI_Trigger_Rising_Falling：设置输入中断线为下降沿和上升沿触发。

EXTI_LineCmd 用来定义待配置中断线的使能状态，其取值为 ENABLE 或者 DISABLE。

例如使能外部中断 12 和 14、下降沿触发：

```
EXTI_InitTypeDef EXTI_InitStructure;
EXTI_InitStructure.EXTI_Line = EXTI_Line12 | EXTI_Line14;
EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling;
EXTI_InitStructure.EXTI_LineCmd = ENABLE;
EXTI_Init(&EXTI_InitStructure);
```

其中，EXIT_Line 设置初始化的外部中断线；EXTI_Mode 设置外部中断模式，取值为 EXTI_Mode_Interrupt 表示产生中断，取值为 EXTI_Mode_Event 表示产生事件；EXTI_Trigger 设置外部中断触发方式，取值为 EXTI_Trigger_Rising 表示上升沿触发，取值为 EXTI_Trigger_Falling 表示下降沿触发，取值为 EXTI_Trigger_Rising_Falling 表示双沿触发；EXTI_LineCmd 设置外部中断线使能。

2. 函数 EXTI_GetFlagStatus

功能描述：检查指定的 EXTI 中断线是否挂起。

函数原型：FlagStatus EXTI_GetFlagStatus(uint32_t EXTI_Line)。

输入参数 EXTI_Line，待读取的 EXTI 中断线名称。

返回值：EXTI_Line 的状态(SET 或者 RESET)。

例如：

```
FlagStatus EXTIStatus;
EXTIStatus = EXTI_GetFlagStatus(EXTI_Line8);
```

3. 函数 EXTI_ClearFlag

功能描述：清除 EXTI 中断线挂起标志位。

函数原型：void EXTI_ClearFlag(uint32_t EXTI_Line)。

输入参数 EXTI_Line，待清除的 EXTI 中断线。

例：

```
EXTI_ClearFlag(EXTI_Line2);
```

4. 函数 EXTI_GetITStatus

功能描述：检查指定的外部中断线上的触发请求是否发生。

函数原型：ITStatus EXTI_GetITStatus(uint32_t EXTI_Line)。

输入参数 EXTI_Line：待检查 EXTI 线路。

返回值：EXTI_Line 的状态(SET 或者 RESET)。

例：

```
EXTIStatus = EXTI_GetITStatus(EXTI_Line8);
```

5. 函数 EXTI_ClearITPendingBit

功能描述：用于清除指定的外部中断线上的中断挂起位，使得在中断发生后进入中断服务程序时，可以继续响应中断请求。

函数原型：void EXTI_ClearITPendingBit(uint32_t EXTI_Line)。

输入参数 EXTI_Line：待清除 EXTI 中断线。

例：

```
EXTI_ClearITpendingBit(EXTI_Line2);
```

6. 函数名 SYSCFG_EXTILineConfig

功能描述：选择 GPIO 引脚作为中断输入，该函数不属于 EXTI 控制器库函数。

函数原型：SYSCFG_EXTILineConfig(uint8_t EXTI_PortSourceGPIOx, uint8_t EXTI_PinSourcex)。

输入参数 1：GPIO 引脚 EXTI_PortSourceGPIOx，x 的取值范围为 A～H。

输入参数 2：配置的中断通 EXTI_PinSourcex，x 取值范围为 0～15。

例如，将 EXTI0 连接到 PA0 引脚：

```
SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOA, EXTI_PinSource0);
```

6.8 中断案例

使用 I/O 口外部中断的一般步骤为：

(1) 初始化 I/O 口为输入。

(2) 开启 I/O 口复用时钟，设置 I/O 口与中断线的映射关系。

(3) 初始化线上中断，设置触发条件等。

(4) 配置中断分组(NVIC)，并使能中断。

(5) 编写中断服务函数。

【例 6-4】 外部中断配置方法：

```
EXTI_InitTypeDef EXTI_InitStructure;
EXTI_InitStructure.EXTI_Line=EXTI_Line4;                    //中断线的标号 EXTI_Line0~EXTI_Line15
EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;         //中断模式:事件、中断
EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling;     //触发方式
EXTI_InitStructure.EXTI_LineCmd = ENABLE;
EXTI_Init(&EXTI_InitStructure);
//涉及中断要设置 NVIC 中断优先级
NVIC_InitTypeDef NVIC_InitStructure;
NVIC_InitStructure.NVIC_IRQChannel = EXTI2_IRQn;            //使能按键外部中断通道
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0x02; //抢占优先级 2
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0x02;       //子优先级 2
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;             //使能外部中断通道
NVIC_Init(&NVIC_InitStructure);
```

在 NVIC 初始化前，我们还需要调用 NVIC_PriorityGroupConfig(NVIC_PriorityGroup_x)配置优先级分组。

配置完中断优先级之后，接着我们要做的就是编写中断服务函数。中断服务函数的名字是在.s 的汇编文件中事先有定义。STM32 的 I/O 口外部中断函数只有 6 个，分别为：

```
EXTI0_IRQHandler
EXTI1_IRQHandler
EXTI2_IRQHandler
EXTI3_IRQHandler
EXTI4_IRQHandler
EXTI9_5_IRQHandler
EXTI15_10_IRQHandler
```

即中断线 0～4 每个中断线对应一个中断函数，中断线 5～9 共用中断函数 EXTI9_5_IRQHandler，中断线 10～15 共用中断函数 EXTI15_10_IRQHandler。

在编写中断服务函数的时候会经常使用到两个函数：

(1) 判断某个中断线上的中断是否发生(标志位是否置位)：

ITStatus EXTI_GetITStatus(uint32_t EXTI_Line)；这个函数一般使用在中断服务函数的开头判断中断是否发生。

(2) 清除某个中断线上的中断标志位：

void EXTI_ClearITPendingBit(uint32_t EXTI_Line)；这个函数一般应用在中断服务函数结束之前，清除中断标志位。

常用的中断服务函数格式为：

```
void EXTI2_IRQHandler(void)
{
```

```
    if(EXTI_GetITStatus(EXTI_Linex) != RESET)              //判断 LINEx 线上的中断是否发生
    {
        中断逻辑…
        EXTI_ClearITPendingBit(EXTI_Line3);                //清除 LINEx 上的中断标志位
    }
}
```

【例 6-5】 通过连接到 PA0 口的按键产生中断,来切换连接到 PB7 口的 LED 灯显示。

```
#include "stm32l1xx.h"
#include "stm32l1xx_gpio.h"
int main(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;                   //通用输入输出的初始化的结构体
    EXTI_InitTypeDef EXTI_InitStructure;                   //外部中断控制器的初始化的结构体
    NVIC_InitTypeDef NVIC_InitStructure;                   //嵌套向量中断控制器的初始化的结构体
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOB, ENABLE); //使能 GPIOB
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_7;           //设置 PB7,用于连接 LED 灯
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;       //设置为输出模式
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;      //设置输出模式下的推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_40MHz;   //设置最大速度为 40MHz
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;    //设置无上拉
    GPIO_Init(GPIOB, &GPIO_InitStructure);              //初始化 GPIOB
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA, ENABLE); //使能 GPIOA
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG,ENABLE);//使能 SYSCFG
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;           //设置 PA0,用于产生按键
                                                        //输入
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;        //设置为输入模式
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;    //设置无上拉
    GPIO_Init(GPIOA, &GPIO_InitStructure);              //初始化 GPIOA
    //连接 EXTI 线和 GPIO 引脚(根据输入输出口外部中断映射)
    SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOA,EXTI_PinSource0);  //EXTI 线 0
    EXTI_InitStructure.EXTI_Line = EXTI_Line0;
    EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;     //配置外部中断模式
    EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising;  //上升沿触发方式
    EXTI_InitStructure.EXTI_LineCmd = ENABLE;               //使能外部中断线 0
    EXTI_Init(&EXTI_InitStructure);                         //初始化外部中断控制
                                                            //器 EXTI
    NVIC_InitStructure.NVIC_IRQChannel = EXTI0_IRQn;        //NVIC 中断源
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0x0F; //抢占优先级 15
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0x0F;   //设置从优先级 15
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;         //使能中断通道
    NVIC_Init(&NVIC_InitStructure);                         //初始化嵌套向量中断控
                                                            //制器
```

```
    while (1);
}
void Delay(__IO uint32_t nCount)
{
  while(nCount--);
}
```

文件 stm32l1xx_it.c 中的中断服务程序,代码如下:

```
void EXTI0_IRQHandler(void)
{
  //检测外部中断线 0上的触发请求
  if(EXTI_GetITStatus(EXTI_Line0) != RESET)
  {
    //切换 LED1状态
    GPIO_ToggleBits(GPIOB, GPIO_Pin_7);
    //清除 EXTI线 0上的挂起位
    EXTI_ClearITPendingBit(EXTI_Line0);
  }
}
```

【例 6-6】 全局开关中断。

按键按下 1次,关中断,再次按下,开中断,代码如下:

```
while (1)
{   //判断按键是否按下弹起
    while(GPIO_ReadInputDataBit(GPIOA,GPIO_Pin_0) ==RESET);
    while(GPIO_ReadInputDataBit(GPIOA,GPIO_Pin_0) !=RESET);
    __disable_irq();                                    //关闭全局中断
    GPIO_SetBits(GPIOB,GPIO_Pin_7);                     //亮灯
    //判断第二次按键是否按下弹起
    while(GPIO_ReadInputDataBit(GPIOA,GPIO_Pin_0) =RESET);
    while(GPIO_ReadInputDataBit(GPIOA,GPIO_Pin_0) !=RESET);
    __enable_irq();                                     //开全局中断
    GPIO_ResetBits(GPIOB,GPIO_Pin_7);                   //灭灯
}
```

Chapter 7

第7章 定 时 器

【导读】 定时器是嵌入式系统中使用最频繁的部件，本章首先介绍定时器(Timer)的基本组成和工作原理，然后对 Cortex-M3 内部定时器和 STM32L152 外围定时器分别作了介绍。内部定时器 SysTick 功能简单，适合驱动操作系统，方便不同厂家微控制器之间的移植；外围定时器包括基本定时器和通用定时器，通用定时器功能较为复杂，除了定时外，还具有输入捕获、输出比较、PWM 产生等功能，多个硬件定时器之间还可以进行级联。STM32L152 的 8 个外围定时器的寄存器有所区别，本章对寄存器定义统一进行了描述，对特殊部分进行了标注，介绍了常用寄存器的各个域的功能，以及 CMSIS 提供的典型寄存器操作库函数，最后以定时中断、比较输出、输入捕获和 PWM 为例介绍了定时器使用方法。

7.1 定时器原理概述

定时器是嵌入式系统中最为常用的一个功能模块，定时器为应用系统提供延迟、定时中断、捕获输入、控制 PWM 输出等一系列功能，也是驱动操作系统运行的关键硬件模块。

我们之前介绍 GPIO 时使用了 Delay 函数，通过一个 for 循环来实现延时。Delay 函数的执行需要 CPU 参与，即 CPU 在 Delay 期间处于运行状态且不能执行其他程序，对于嵌入式系统，这种实现功耗较高，系统性能受到影响。定时器是一个独立于 CPU 的硬件，采用中断方式和 CPU 进行交互，可以有效提高嵌入式微控制器整体性能。定时器的基本原理是通过一个计数器自动计数，当计数到某个特定值时，触发一个中断，计数的周期和计数值用于确定计数器持续的时间。

定时器的基本原理如图 7-1 所示，由预分频器、自动重装载寄存器、计数器、比较寄存器和中断输出单元构成。

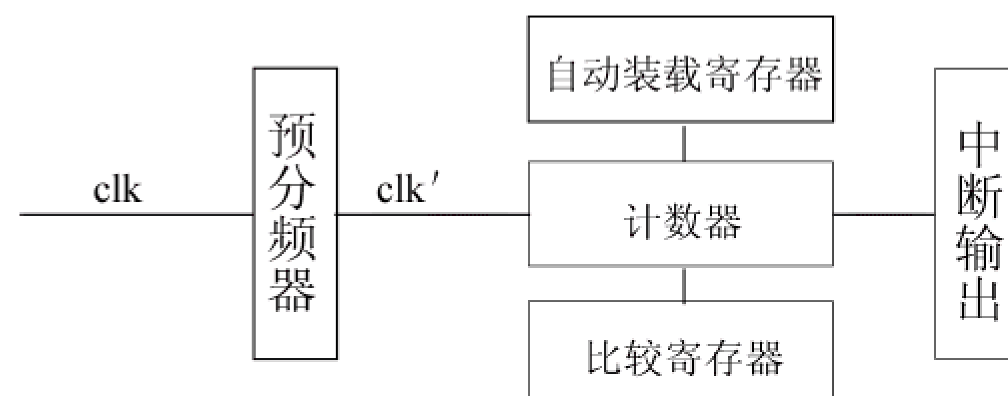


图 7-1 定时器基本结构

计数器在时钟驱动下工作,每个时钟脉冲计数器自动加1或减1,时钟的频率由预分频器决定,预分频器可以将定时器的输入时钟进行1,2,4,8等分频,这样一个计数值可以表示更长的时间间隔。自动重装载寄存器是用户可配置的,计数器在自加模式下,从0开始计数,当计数值和自动重装载寄存器值相同时,产生一个中断,并将计数器值清零;当计数器工作在自减模式时,计数器的初值为自动重装载寄存器的值,当计数器递减到0时产生一个中断,并将计数器的初值重新赋为自动重装载寄存器值。如果没有配置自动重装载寄存器,则计数器默认的装载值为计数器能表示的最大值(溢出值)。除了计数器时间到产生中断外,有些定时器还支持比较寄存器中断输出方式,即当计数器的值和比较寄存器值相同时产生一个中断,用户可以不断设置比较寄存器的值在计数器溢出前产生多次中断。

定时器的计时长度等于计数值/预分频后的时钟频率,一般定时器的计数器为8bit、16bit和32bit,我们称为8位定时器、16位定时器和32位定时器。定时器位数越长,计时长度越长;预分频越小,一个时间脉冲的长度越短,表达的时间越精确,但定时器能表示的最长时间间隔越短。因此在使用定时器时,需要根据需求调整预分频参数和定时器长度。

在嵌入式系统中,我们需要控制多个时间值或者控制不同精度的定时,一般嵌入式微控制器会提供多个硬件定时器,这些定时器可以并行使用,但软件系统用到的定时器根据应用不同可能会有很多,因此硬件定时器不够时我们需要基于一个硬件定时器实现多个软件定时器,以满足系统需求。

【思考题: 如何通过一个硬件定时器实现 n 个软件定时器?**】**

为适应不同嵌入式系统的需求,STM32L152提供了丰富的定时器功能,结构上也比图7-1所示的定时器基本结构复杂,主要包括SysTick定时器、通用定时器和基本定时器(不同于STM32F系列,L系列没有高级定时器)。此外,看门狗定时器和实时时钟RTC也可作为普通定时使用。

SysTick是一个24位的倒计数定时器,当计到0时,将自动装载定时初值,其主要用于为操作系统提供一个硬件的滴答中断,进行进程调度。SysTick定时器由Cortex-M3定义,存在于NVIC控制器中,这样便于同是Cortex-M3内核、不同厂家的嵌入式处理器进行系统移植。在无操作系统的嵌入式系统中,该计时器可以作为一个普通定时器使用。

STM32L152外围控制器提供了通用定时器和基本定时器,两类定时器的功能不同,通用定时器除了基本定时器功能外,还有向上/向下计数、PWM、输出比较、输入捕获等功能,不同的通用定时器在捕获通道数量、编码器等功能上也有所区别。STM32L152提供的定时器比较如表7-1所示。

表7-1　STM32L152外围定时器比较

定　时　器	计数模式	计数器长度	捕获/比较通道数量	其他功能
通用定时器 TIM2、TIM3、TIM4	向上、向下、向上/向下	16位	4	外部时钟触发和定时器同步,支持编码器
通用定时器 TIM9	向上	16位	2	外部时钟触发和定时器同步

续表

定　时　器	计数模式	计数器长度	捕获/比较通道数量	其 他 功 能
通用定时器 TIM10、TIM11	向上	16 位	1	无
基本定时器 TIM6、TIM7	向上	16 位	0	DAC 触发

7.2　内部定时器 SysTick

SysTick 定时器是 Cortex-M3 内核自带的定时器，该定时器的时钟源一般采用内核时钟，因此，采用 SysTick 使得软件在不同的 Cortex-M3 处理器上的移植更加方便。由于内核时钟频率较高，SysTick 定时器具有较高精度，通常可用于精确计时和测量。

SysTick 定时器在 NVIC 中，Cortex-M3 为该定时器设有 SYSTICK 异常。SysTick 是一个 24 位倒计时定时器，最大可计数 2^{24}，计数值保存在 STK_VAL 寄存器中，每过一个时钟周期，STK_VAL 的值减 1，当减到 0 时，触发 SYSTICK 异常，同时硬件自动把重装载寄存器 STK_LOAD 中的数据加载到 STK_VAL，重新开始向下计数。

7.2.1　SysTick 寄存器

SysTick 定时器的控制涉及四个寄存器，如表 7-2 所示。

表 7-2　SysTick 定时器寄存器及其功能

寄存器名	寄存器地址	读 写 权 限	功　　能
SYST_CSR	0xE000E010	RW	SysTick 控制和状态寄存器
SYST_RVR	0xE000E014	RW	SysTick 重装载寄存器
SYST_CVR	0xE000E018	RW	SysTick 计数值寄存器
SYST_CALIB	0xE000E01C	RO	SysTick 校准寄存器

SysTick 定时器的主要寄存器的具体定义如下。

1）SysTick 控制和状态寄存器 SYST_CSR

SYST_CSR 是一个 32 位寄存器，如图 7-2 所示，其有效域有四个。

其中 bit[16]为 COUNTFLAG 计数标志位，用来表示 SysTick 的计数值是否已经数到了 0，如果已经到 0，则该位被置 1，如果读取该位，该位将被自动清零。

bit[2]为 CLKSOURCE，表明 SysTick 时钟源，该位为 1 时表示采用处理器主时钟 HCLK 作为时钟源，0 表示采用 HCLK/8 作为时钟源。

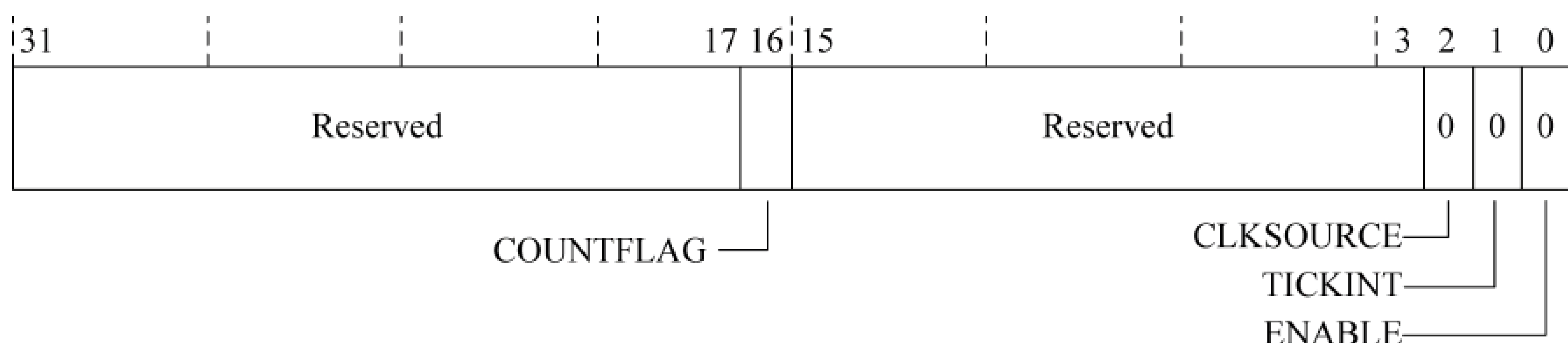


图 7-2　SysTick 控制和状态寄存器

Bit[1]为 TICKINT 中断使能位，该位为 1 表示 SYST_CSR 计数到 0 时产生中断，0 表示不产生中断。该位为 0 时，可以通过监测 COUNTFLAG 判断 SYST_CSR 是否计数到 0。

Bit[0]为 ENABLE 使能位，该位为 1 表示启用 SysTick 定时器，0 表示关闭 SysTick 定时器。

2）SysTick 重载寄存器 SYST_RVR

SYST_RVR 是一个 32 为寄存器，如图 7-3 所示，有效位为低 24 位 bits[23：0]，保存 SYST_CVR 计数到 0 时加载到 SYST_CVR 寄存器的计数值。RELOAD 的取值范围为 0x00000001～0x00FFFFFF，要产生 N 个时钟周期的中断，RELOAD 设为 N－1。



图 7-3　SysTick 重载寄存器

3）SysTick 当前计数值寄存器 SYST_CVR

SYST_CVR 中存放 SysTick 定时器当前的计数值，如图 7-4 所示有效位 24 位，读取该寄存器时返回 CURRENT 的值，写该寄存器，CURRENT 将被置 0，同时 SYST_CSR 的 COUNTFLAG 位也将被置 0。



图 7-4　SysTick 计数值寄存器

4）SysTick 校准值寄存器 SYST_CALIB

SYST_CALIB 用于存放 SysTick 定时器校准值，即 4M 时钟 1ms 时间间隔的计数值，默认为 4000。

SysTick 定时器的配置流程和工作过程如下：

- 配置 SYST_RVR，设定定时器的计数周期数。
- 清空 SYST_CVR 寄存器的计数值。
- 配置 SYST_CSR 寄存器，设定 SysTick 计数器的时钟源，是否启用中断。
- 配置 SYST_CSR 寄存器，使能 ENABLE，启动定时器。

当 ENABLE 为 1 时，定时器将 SYST_RVR 寄存器的 RELOAD 值加载到 SYST_CVR 并开始向下计数，当计数到 0 时，将 SYST_CSR 寄存器的 COUNTFLAG 位置 1 并根据 TICKINT 的值确定是否产生中断请求，然后将 SYST_RVR 寄存器的 RELOAD 值再次加载到 SYST_CVR 寄存器，启动向下计数。

如果允许中断，调用中断处理程序中；如不启用中断，可通过不断读取 SYST_CSR 寄存器的 COUNTFLAG 标志位判断是否计时至零。

7.2.2 SysTick 定时器库函数

为了便于对 SysTick 定时器进行操作，CMSIS 提供了 SysTick 定时器的寄存器定义和库函数。

SysTick 定时器的寄存器结构体定义在 core_cm3.h 头文件中，我们可以通过结构体类型 SysTick_Type 定义变量对 SysTick 定时器进行控制。

```
typedef struct {
__IO uint32_t CTRL;                    //SysTick 控制和状态寄存器
__IO uint32_t LOAD;                    //SysTick 重载寄存器
__IO uint32_t VAL;                     //SysTick 计数值寄存器
__IO uint32_t CALIB;                   //SysTick 校准值寄存器
} SysTick_Type;
```

【例 7-1】 用 SysTick 定时器产生任意大小 T 的精确延迟，以微秒为单位。

分析：我们首先选取 SysTick 定时器的时钟源，HCLK 或 HCLK/8，STM32L152 的最高频率 HCLK 为 32MHz，因此以 HCLK 为时钟源，一个时钟周期为 1/32 微秒，因此对于任意大小时间 T，其重载寄存器的值应为 32 * T。定义一个 SysTick_Type 类型的变量配置 SysTick 定时器的寄存器，通过查询 CTRL 寄存器的 COUNTFLAG 判断计时是否到，代码如下：

```
SysTick_Type SysTick;                  //定义一个 SysTick类型的定时器变量
void Delay_us(uint32_t n)              //单位为 μs
{
  SysTick->LOAD=32*n;                  //配置重载寄存器值
  SysTick->CTRL=0x00000005;            //时钟源 HCLK(32M),打开定时器
  while(!(SysTick->CTRL&0x00010000));  //等待计数到 0
  SysTick->CTRL=0x00000004;            //关闭定时器
}
```

CMSIS 为 SysTick 定时器提供了两个操作函数，如表 7-3 所示，可直接调用方便使用，其中 SysTick_CLKSourceConfig 定义在 misc.c 中，SysTick_Config 定义在 core_cm3.h 中。

表 7-3　SysTick 库函数

函数名称	函数功能
SysTick_CLKSourceConfig	配置 SysTick 定时器时钟源
SysTick_Config	初始化并开启 SysTick 计数器及中断

1）SysTick_CLKSourceConfig 函数

函数原型：void SysTick_CLKSourceConfig(uint32_t SysTick_CLKSource)。

输入参数：SysTick_CLKSource，用于表明 SysTick 定时器的时钟源，其取值为：

- SysTick_CLKSource_HCLK_Div8，HCLK/8 作为 SysTick 时钟源；
- SysTick_CLKSource_HCLK，HCLK 作为 SysTick 时钟源；

示例：

```
SysTick_CLKSourceConfig(SysTick_CLKSource_HCLK);
//设置 AHB 时钟为 SysTick 时钟源，32MHz
```

2）SysTick_Config 函数

函数原型：uint32_t SysTick_Config(uint32_t ticks)。

输入参数：ticks，两次中断间的间隔数值。

函数返回值，0 表示成功，1 表示失败。

SysTick_Config()函数将参数 ticks 写到 Systick 的重载寄存器中，配置 SysTick 中断优先级为最低(0x0F)，清除 Systick 当前计数值寄存器，配置时钟源为 HCLK，使能中断并启动计数。定时器时间(秒)＝ticks/HCLK。具体实现如下：

```
uint32_t SysTick_Config(uint32_t ticks)
{
  //检查 ticks，如果 ticks 超过 24 位，返回失败
  if (ticks > SysTick_LOAD_RELOAD_Msk)   return (1);
  //配置装载寄存器，SysTick_LOAD_RELOAD_Msk= 0xFFFFFFFFul
  SysTick->LOAD  = (ticks & SysTick_LOAD_RELOAD_Msk) - 1;
  //配置 SYSTICK 中断优先级，设为最低 0xF
  NVIC_SetPriority(SysTick_IRQn, (1<<__NVIC_PRIO_BITS) - 1);
  SysTick->VAL = 0;                        //计数值清零
  //配置控制寄存器，CLKSOURCE= HCLK，中断使能，定时器启动
  SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk | SysTick_CTRL_TICKINT_Msk
                  | SysTick_CTRL_ENABLE_Msk;
  return (0);                              //初始化成功返回 0
}
```

中断发生后，调用 SysTick 的中断处理程序 SysTick_Handler，在文件 stm32l1xx_it.c，SysTick_Handler 的定义如下：

```
void SysTick_Handler(void)                 //systick 中断处理函数
```

```
{
}
```

如果需要更改时钟源，在 SysTick_Config 函数后调用 SysTick_CLKSource Config(SysTick_CLKSource_HCLK_Div8)；如果需要更改 SysTick 中断优先级，在 SysTick_Config 函数后调用 NVIC_SetPriority(SysTick_IRQn,. number.)。

示例：

```
SysTick_Config(320)                          //设置 SysTick定时器时间为 10μs
```

7.2.3 SysTick 定时器应用例程

【例 7-2】 启动用 SysTick 定时器中断，实现 LED 灯控制。

分析：对例 5-1 的 LED 控制程序进行修改，利用 SysTick_Config 配置 SysTick 定时器，在 SysTick 中断函数中设置状态变量 flag 进行翻转，main 函数中根据 flag 对灯进行控制，代码如下：

```
#include "stm32l1xx.h"
#include "stm32L1xx_gpio.h"
int flag=0;
GPIO_InitTypeDef          GPIO_InitStructure;
int main(void)
{
   //配置 PB7 为推挽输出模式
   RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOB, ENABLE);
   GPIO_InitStructure.GPIO_Pin=GREEN_LED;
   GPIO_InitStructure.GPIO_Mode=GPIO_Mode_OUT;
   GPIO_InitStructure.GPIO_OType=GPIO_OType_PP;
   GPIO_InitStructure.GPIO_Speed=GPIO_Speed_40MHz;
   GPIO_InitStructure.GPIO_PuPd=GPIO_PuPd_NOPULL;
   GPIO_Init(GPIOB, &GPIO_InitStructure);
   SysTick_Config(500*32*1000)               //配置定时器 500ms
   while (1)
   {
       if(flag)                              //根据 flag对灯进行控制
          GPIO_SetBits(GPIOB, GPIO_Pin_7);
       else
          GPIO_ResetBits(GPIOB, GPIO_Pin_7);
   }
}
```

在文件 stm32l1xx_it.c 中的 SysTick_Handler 中断服务程序中，添加如下代码：

```
extern int flag;
void SysTick_Handler(void)                    //SysTick中断处理函数
{
  if(flag)
      flag = 0;
  else
    flag = 1;
}
```

7.3　外围定时器基本概念

除了 Cortex-M3 内核带的 SysTick 定时器外，STM32L152 还在外围总线上提供了多个硬件定时器，这些硬件定时器命名为 TIMx(x=2,3,4,6,7,9,10,11)。外围定时器 TIMx 比 SysTick 定时器功能更为强大，相比图 7-1，结构也较为复杂，由时基单元和捕获比较单元组成。

1. 时基单元

时基单元是定时器的基本单元，包括预分频器、计时器和自动装载寄存器，完成最基本的计时功能。时基单元带有一个自动重装载的累加计数器，计数器的时钟通过一个预分频器得到。软件可以读写计数器、自动重装载寄存器和预分频寄存器，计数器运行时也可以进行读写操作。

时基单元的主要寄存器包括：

- 计数器寄存器(TIMx_CNT)。
- 预分频寄存器(TIMx_PSC)。
- 自动重装载寄存器(TIMx_ARR)。

定时器时基单元的配置与定时器驱动时钟、计数方式的选择有密切关系，以下对 STM32L152 的定时器时钟源和计数方式进行简要介绍。

1) 时钟源

定时器的工作时钟来源于预分频器分频后的时钟 CK_CNT，预分频器的输入时钟为 CK_PSC，CK_PSC 的时钟来源包括以下四种，每个 TIMx 支持的时钟源也不同：

- 内部时钟(CK_INT)。
- 外部时钟模式 1：外部输入脚(TIx)。
- 外部时钟模式 2：外部触发输入(ETR)。
- 内部定时器触发输入(ITRx)。

(1) 内部时钟源(CK_INT)：即定时器采用 CK_INT 作为时钟源，CK_INT 由微控制器的总线 APB1 或 APB2 的时钟提供，定时器的时钟不是直接来自 APB1 或 APB2，而是来自于 APB1 或 APB2 的一个倍频器，当 APB1 或 APB2 的预分频系数为 1 时，这个倍频器不

起作用，定时器的时钟频率等于 APB1 或 APB2 的频率；当 APB1 或 APB2 的预分频系数为其他数值（2、4、8 或 16）时，定时器的时钟频率等于 APB1 或 APB2 的频率两倍。如图 7-5 所示，当 APB1 总线被设置为 32MHz 时（预分频为 1），CK_INT 的时钟为 32MHz，当 APB1 总线被设置为 16MHz 时，此时倍频器起作用，CK_INT 的时钟频率被设为 32MHz。

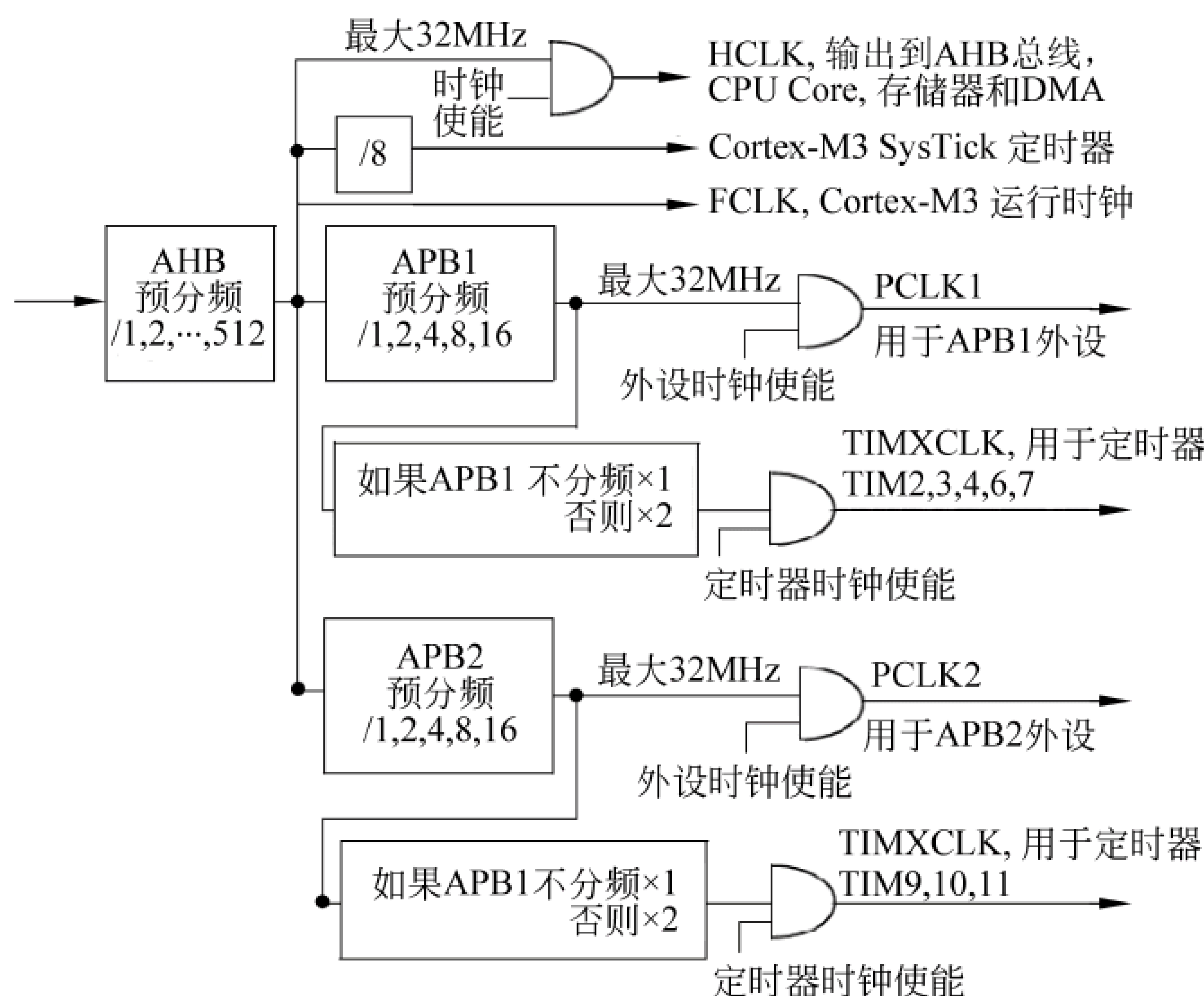


图 7-5　定时器内部时钟源

STM32L152 中，TIM2～TIM7 连接在 APB1 总线，TIM9～TIM11 连接在 APB2 总线，因此在使用不同的 TIMx 时需要配置不同的总线频率。

（2）外部时钟源模式 1 下，时钟来源于 MCU 外部引脚，时钟源选择 TIMx 输入通道 1 或输入通道 2 所对应引脚，定时器在 TIMx 输入通道对应引脚的电平边沿信号作为时钟驱动。

（3）外部时钟源模式 2 下，时钟源来源于 MCU 外部引脚，时钟源为外部触发引脚 ETR，定时器在 ETR 引脚的每一个上升沿或下降沿计数。

（4）内部定时器触发模式下，可以使用一个定时器作为另一个定时器的预分频器时钟输入，每个定时器最多可以有 4 个其他内部定时器触发时钟源，如 TIM4 可以用 TIM10、TIM2、TIM3 和 TIM9 作为内部时钟源，具体参见 STM32L152 参考手册。

基本定时器 TIM6 和 TIM7 的时钟只能由内部时钟提供，但通用定时器 TIM2～TIM5、TIM9～TIM11 可以选择多种时钟源。目前定时器使用中，基本上都是采用内部时钟作为时钟源。

2）计数方式

计数模式包括三种模式：向上计数模式、向下计数模式和中央对齐模式。

（1）向上计数模式：计数器从 0 计数到自动加载值（TIMx_ARR 计数器的值），然后重新从 0 开始计数并且产生一个计数器溢出事件和更新事件，当发生一个更新事件时，所有的

寄存器都被更新,如图 7-6 所示。

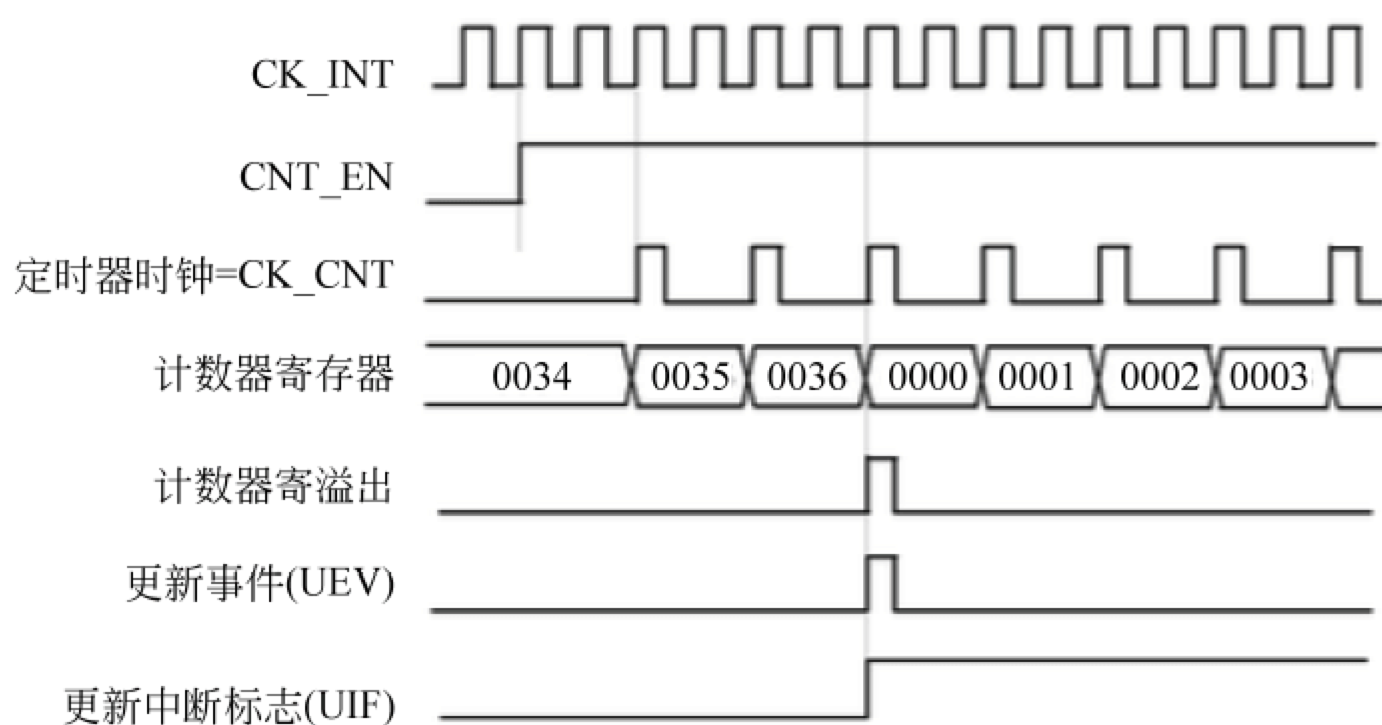


图 7-6 向上计数模式

(2) 向下计数模式:在向下模式中,计数器从自动装入的值(TIMx_ARR 计数器的值)开始向下计数到 0,然后从自动装入的值重新开始,并产生一个计数器向下溢出事件和更新事件,如图 7-7 所示。

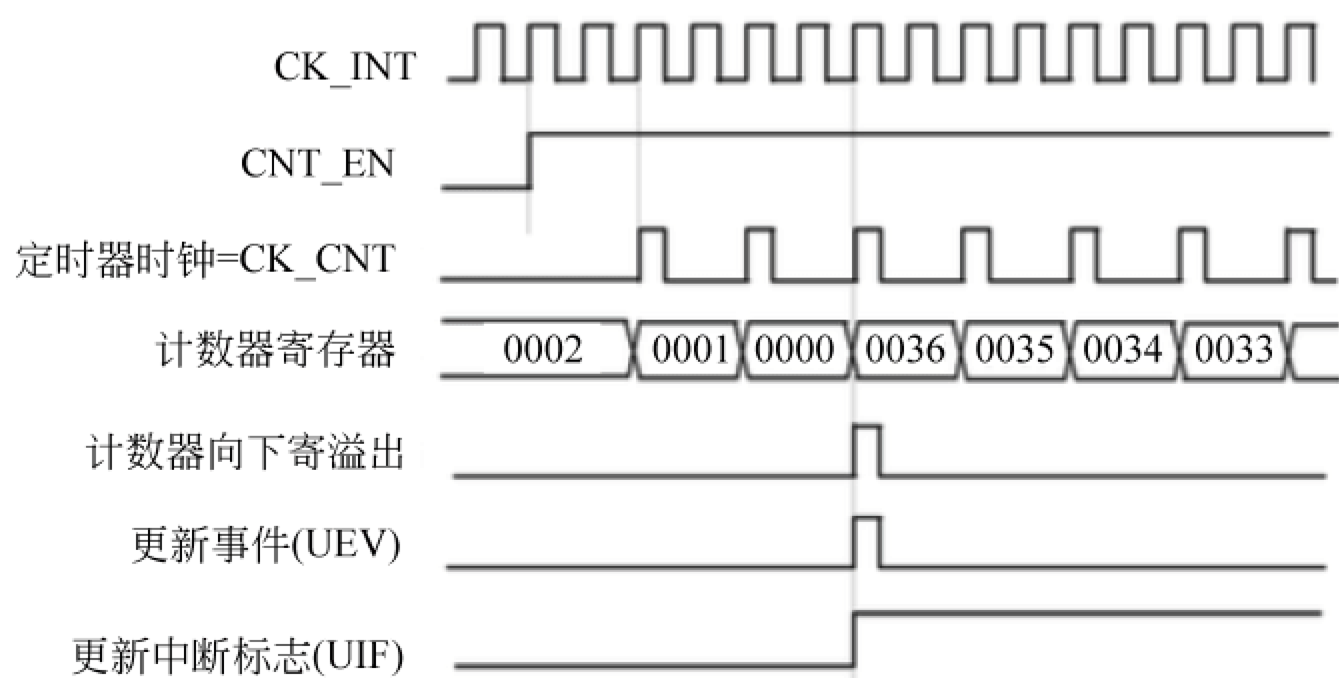


图 7-7 向下计数模式

(3) 中央对齐模式:在中央对齐模式,计数器从 0 开始计数到自动加载值-1,即 TIMx_ARR-1,产生一个计数器上溢事件,然后向下计数到 1 并且产生一个计数器下溢事件,然后再从 0 开始重新计数。计数的方向由硬件寄存器指示,可以在每次计数上溢和每次计数下溢时产生更新事件,然后,计数器重新从 0 开始计数,如图 7-8 所示。

2. 捕获比较单元

捕获比较单元可对输入信号进行捕捉,或根据设定输出不同的信号。

(1) 输入捕获:可以用来捕获外部事件,并为其赋予时间标记以说明此事件的发生时刻。外部事件发生的触发信号由单片机中对应的外部引脚输入,比如按下按键,也可以通过内部单元(如模拟比较器等)来实现。捕获的信号可以设置为上升沿捕获、下降沿捕获、或者上升沿下降沿都捕获,当设置的捕获发生时,微控制器会将计数寄存器的值复制到捕获比较寄存器,当再次捕捉到电平变化时,新的捕获比较寄存器的值减去之前复制的值就是输入信

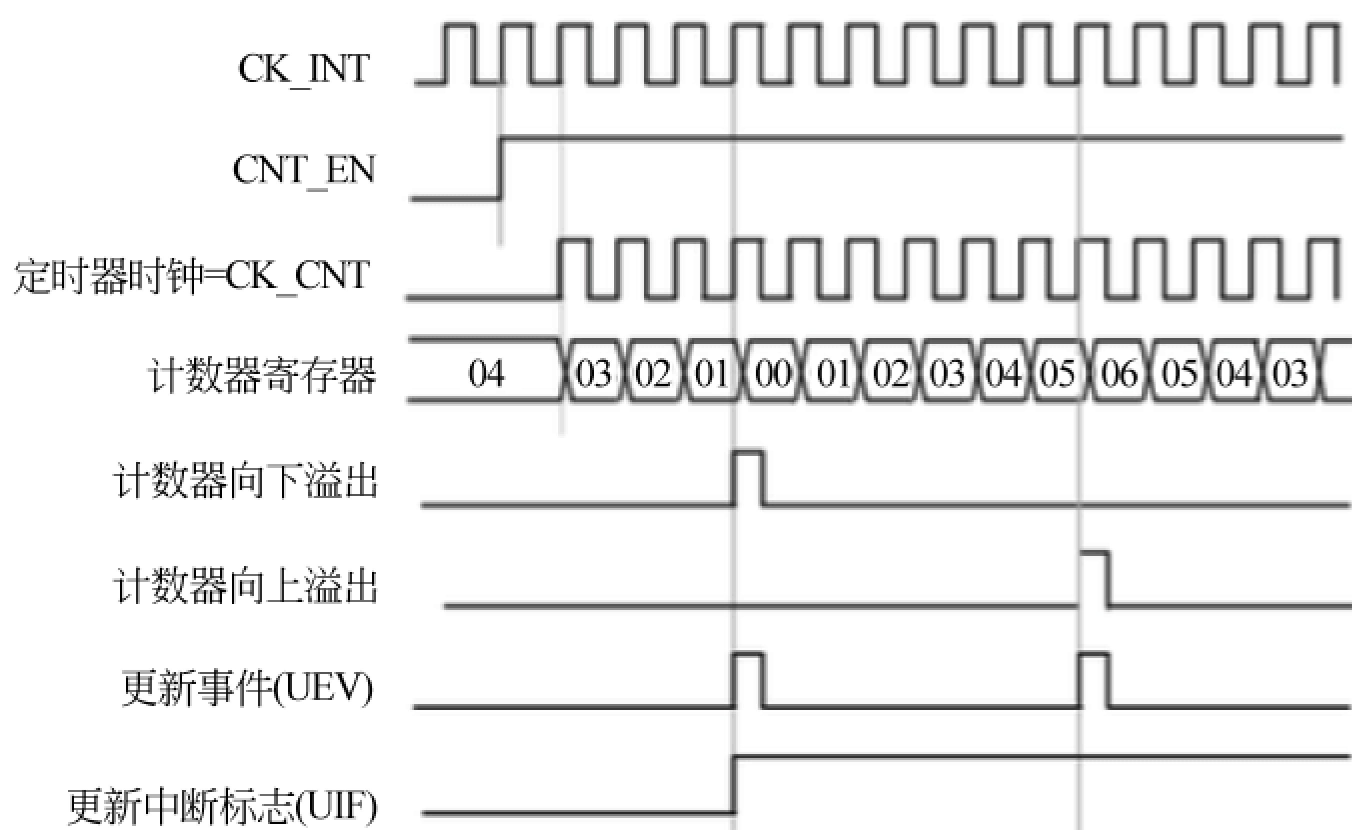


图 7-8　中央计数模式

号的间隔或持续时间。

(2) 输出比较：定时器中计数寄存器在初始化完后会自动的计数，一旦计数寄存器在计数过程中与比较寄存器匹配则会产生匹配事件，此时我们可以产生一个中断，也可以在 GPIO 的端口输出一个电平变化(变低、变高或取反)，用于控制其他外设。

STM32L152 的不同定时器支持的捕获和输出比较通道的数量不同，两通道捕获和输出比较电路原理如图 7-9 所示，GPIO 端口 TIMx_CH1 和 TIMx_CH2 可作为捕获输入源或者比较输出源使用。作为输入捕获时，首先对输入信号进行滤波，生成 TI1F，然后进行上升沿、下降沿检测，生成信号 TI1FP1，选通器可以选择通道 1，也可以选择通道 2 作为捕获信号 IC1，即通道 1 同时连接到了 IC1 和 IC2，这样连接的目的可以用两个捕获通道对于同一个信号的不同边沿进行检测。分频器可根据配置对 IC1 进行分频，分频后的 IC1PS 脉冲边沿被监测到时，将计数器的值保存到捕获比较寄存器，并产生捕获事件。作为比较输出时，捕获比较寄存器与计数器的值进行比较，当发生匹配时，产生信号 OC1REF，通过输出控制在 TIMx_CH1 端口输出高电平或低电平。

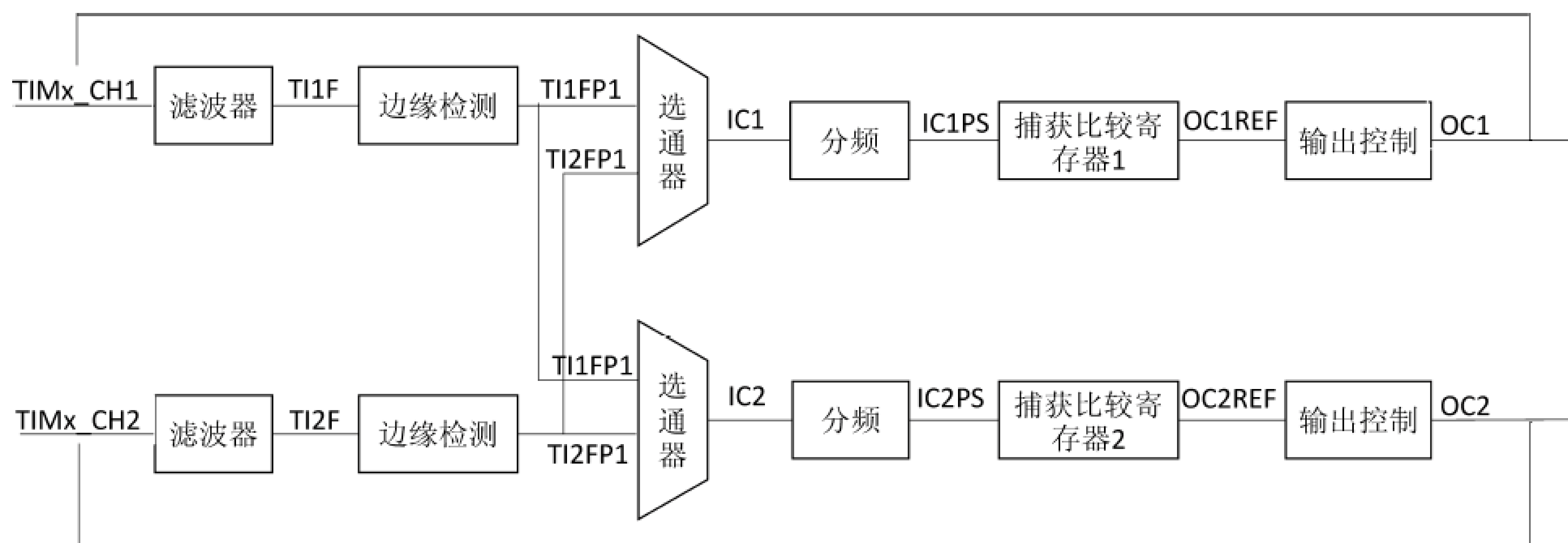


图 7-9　捕获和输出比较原理

3. 定时器工作中的中断和事件

定时器工作中,会在一些关键点触发一些事件,用于管理定时器的状态,比如计数器向上计数模式中,计数器从 0 计数到用户定义的比较值(TIMx_ARR 寄存器的值),然后重新从 0 开始计数并产生一个计数器溢出事件。事件发生时,定时器的相关寄存器会记录事件状态,但并不一定会向 CPU 发起中断请求,只有中断允许被配置的情况下定时器才会发生中断请求。因此定时器事件和中断是不同的概念,定时器中断依赖于定时器事件,但定时器事件发生并不一定产生中断。为了便于软件对硬件定时器的管理,定时器支持软件产生事件的功能,当读写定时器状态寄存器的相关控制域时,可以立即生成一个事件,便于程序控制定时器,也可以通过控制寄存器禁用事件产生。

4. 影子寄存器

在定时器控制中,一些寄存器在电路实现时通常对应有两个寄存器,一个用于用户读写访问,称之为预加载寄存器,一个是用户看不见但用于实际控制,称之为影子寄存器。例如,时基单元的自动重装载寄存器带有影子寄存器,用户对于自动重载寄存器的读写实际上是通过读写预加载寄存器实现的。这样的好处是当定时器正在工作时,读写自动重装载寄存器不会影响原先定时器的工作,所有真正需要起作用的寄存器可以在同一个特定条件(如更新事件产生)触发时才把预加载寄存器的值写入到影子寄存器,以保证多个通道控制的同步性,当然用户也可配置成立即写入影子寄存器的方式。本章以下部分定时器结构图中,凡是带有阴影的寄存器都有影子寄存器。

7.4　基本定时器 TIM6、TIM7

基本定时器包括两个独立的 16 位定时器 TIM6 和 TIM7,可用于一般的定时时钟或作为驱动模数转换的 DAC 输出时钟(内部已连接到 DAC),其定时器结构如图 7-10 所示。

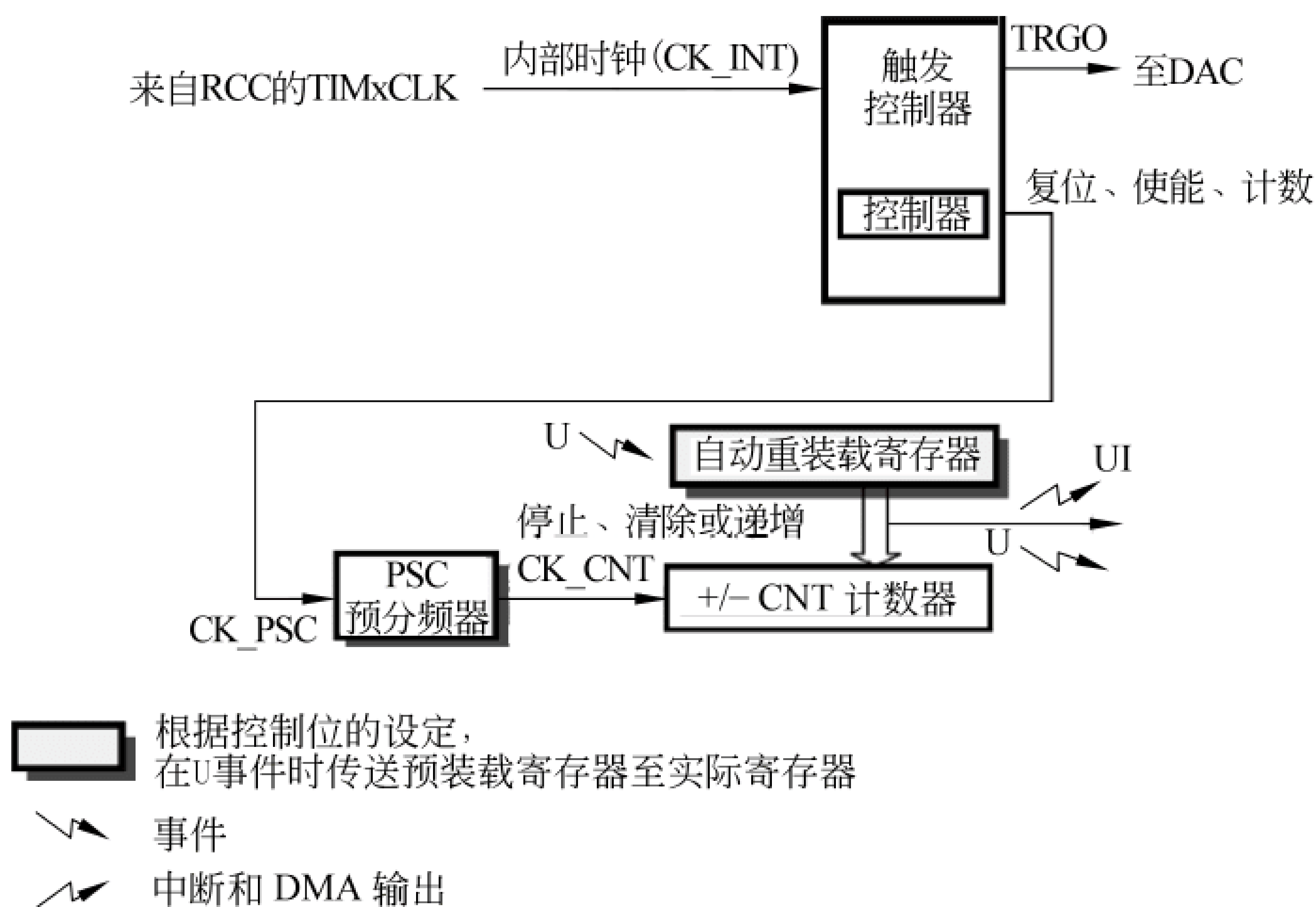


图 7-10　基本定时器结构

基本定时器由控制单元和时基单元组成。控制单元用于配置定时器的参数，获取定时器工作状态，主要由控制寄存器 TIMx_CR1、TIMx_CR2，中断使能寄存器 TIMx_DIER、状态寄存器 TIMx_SR 和事件产生寄存器 TIMx_EGR 组成，在更新事件发生时可以产生中断/DMA 请求。时基单元用于定时器计数控制，主要包括计数寄存器 TIMx_CNT、预分频寄存器 TIMx_PSC 和 自动重装载寄存器 TIMx_ARR。

基本定时器连接在 APB1 总线上，其时钟由内部时钟 CK_INT 提供，CK_INT 不超过总线的最高频率 32MHz，为满足定时器的精度和计时时间长短的要求，可以通过预分频器对总线时钟进行分频。计数器由预分频输出的 CK_CNT 驱动，预分频器是通过一个 16 位寄存器（TIMx_PSC）实现分频，可以以 1～65536 的任意数值对计数器时钟分频。

如图 7-10 所示，TIMx_PSC 预分频寄存器和 TIMx_ARR 带有影子寄存器，在运行过程读写中改变 TIMx_PSC 和 TIMx_ARR 的数值实际改变的是预加载寄存器的值，预加载寄存器的值将在下一个更新事件 UEV 时写入到影子寄存器。TIMx_CR1 寄存器中的自动重装载预加载使能位 ARPE 决定是将写入预加载寄存器的内容立即传送到影子寄存器，还是在更新事件 UEV 时传送到影子寄存器。

如图 7-11 和图 7-12 所示，发生一次更新事件 UEV 时，定时器将设置更新标志位，并将以下寄存器立即更新：

- 传送 TIMx_PSC 预装载值至预分频器的影子寄存器；
- 更新自动重装载影子寄存器为 TIMx_ARR 预装载值。

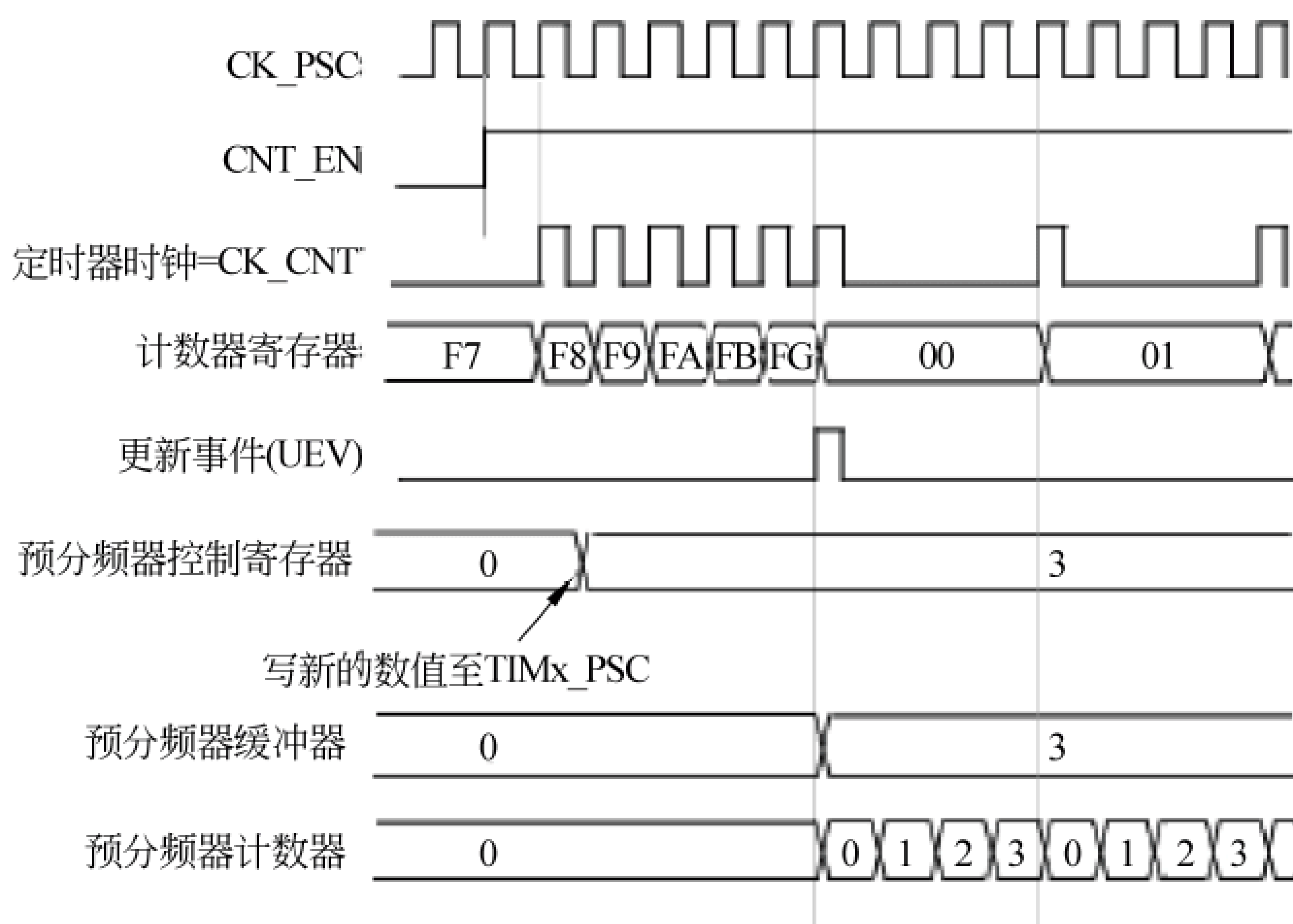


图 7-11　PSC 预装值操作示例

基本定时器的操作流程和工作过程如下：

- 配置预分频寄存器 TIMx_PSC，计数器的时钟频率 CK_CNT 等于 fCK_PSC/（PSC[15：0]＋1）；
- 配置自动重装载寄存器 TIMx_ARR 的计数值；

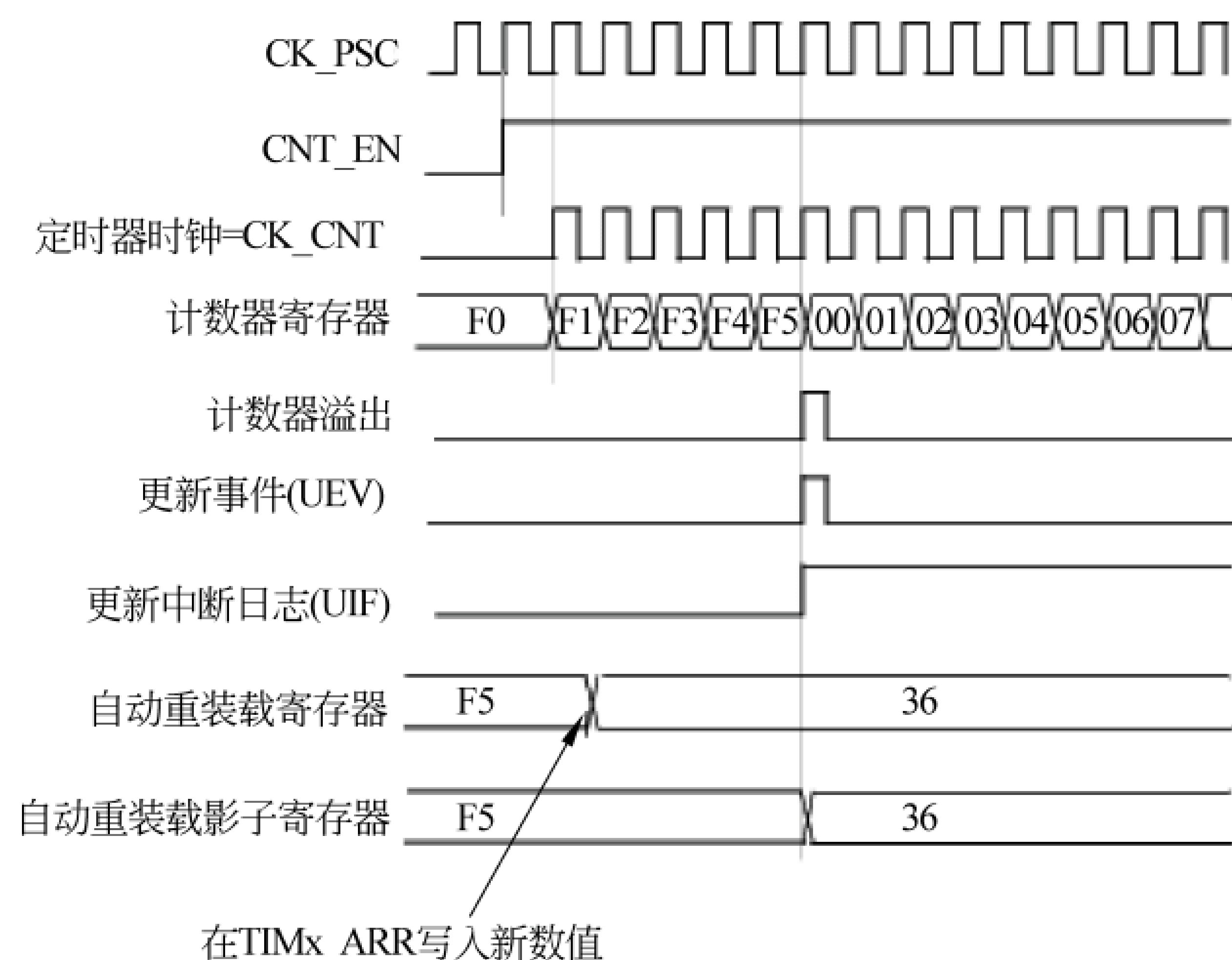


图 7-12　ARR 预装载操作示例

- 配置中断使能寄存器 TIMx_DIER，是否启动 DMA 和中断；
- 配置事件产生寄存器 TIMx_EGR 清空计数器，产生一个软件更新事件，更新所有寄存器配置；
- 配置控制寄存器 TIMx_CR1，设置 CEN 位为 1，启动定时器。

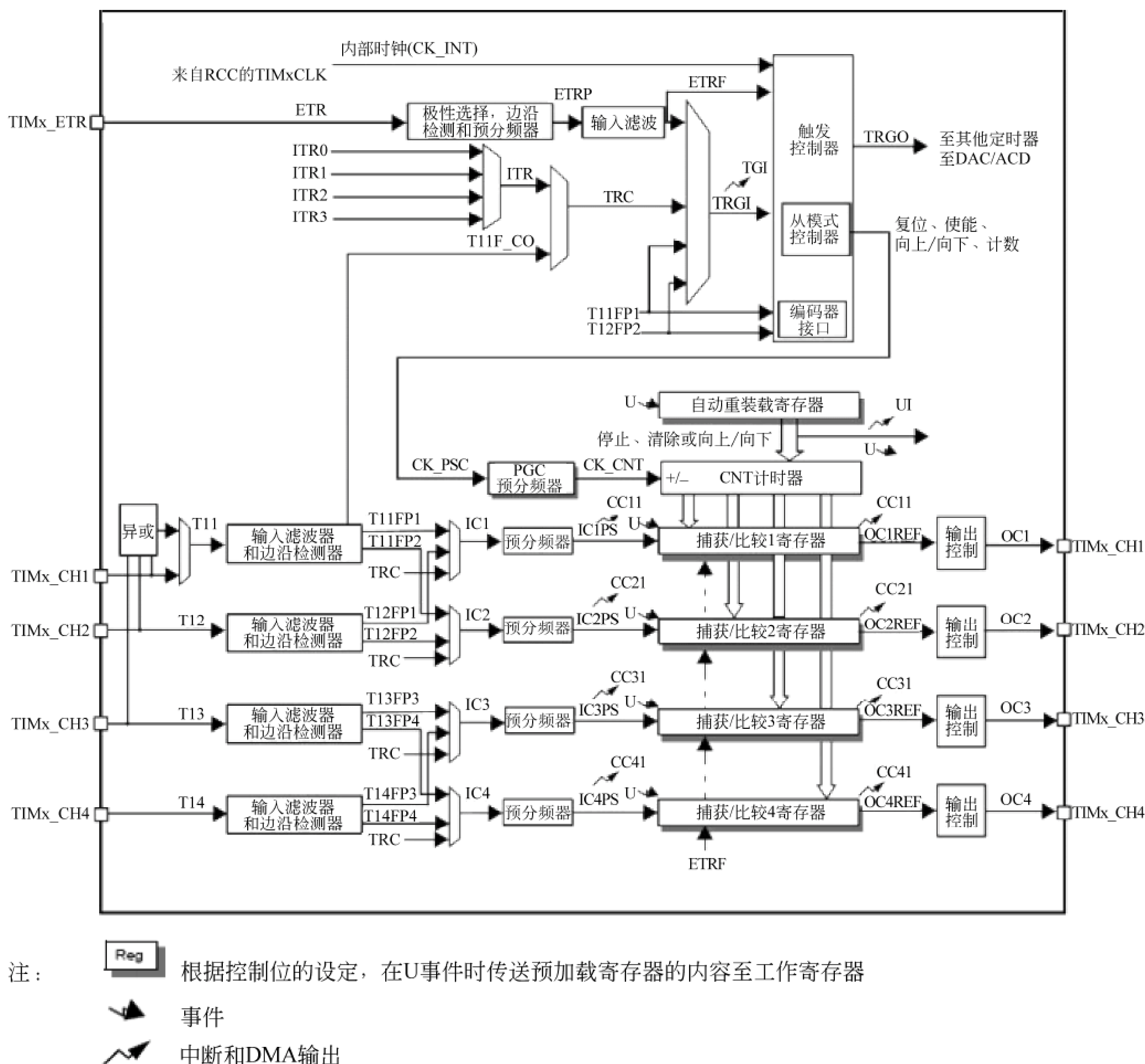
基本定时器只支持向上计数模式，计数器从 0 累加计数到自动重装载数值，产生一个计数器溢出事件，如果 TIMx_CR1 寄存器的 OPM（One Pluse Mode）被置为 1，则停止计数，CEN 被置 0；如果 OPM 被置为 0，则定时器重新从 0 开始计数。

7.5　通用定时器 TIM2～TIM4、TIM9～TIM11

通用定时器除了基本定时器的功能外，还包括向下、向上/向下自动装载计数，1～4 个独立通道用于输入捕获、输出比较、PWM 生成或单脉冲模式输出，可以使用外部信号控制定时器，支持定时器互连，多种中断/DMA 事件产生等功能。

STM32L152 有 6 个 16 位的通用定时器，分为两类，其中 TIM2～TIM4 连接在 APB1 总线上，各有 4 个独立的捕获比较通道，可以和 TIM9、TIM10 以及 TIM11 进行同步互联；TIM9～TIM11 连接在 APB2 总线上，TIM10 和 TIM11 只有 1 个捕获比较通道，TIM9 有 2 个捕获比较通道，可以被 TIM2、TIM3 和 TIM4 同步；同时 TIM9、TIM10 和 TIM11 这三个定时器可以使用 LSE 作为外部时钟源独立于总线时钟工作。

STM32L152 通用定时器的结构如图 7-13 所示，除了控制单元，时基单元外，还增加了时钟源选择、输入滤波和检测、捕获比较以及输出控制等单元。



注：Reg 根据控制位的设定，在U事件时传送预加载寄存器的内容至工作寄存器

事件

中断和DMA输出

图 7-13　通用定时内部结构

7.5.1　通用定时器时基单元

通用定时器的时基单元与基本定时器结构相同，由计数器寄存器 TIMx_CNT、预分频器寄存器 TIMx_PSC 和自动装载寄存器 TIMx_ARR 构成，但其计数模式支持三种模式：向上计数模式、向下计数模式和中央对齐模式。计数模式通过控制寄存器 TIMx_CR1 的计数方向域 DIR 和中央对齐模式域 CMS 进行设置（中央对齐模式下，不能写入 TIMx_CR1 中的 DIR 方向位，DIR 由硬件更新并指示当前的计数方向）。

每次计数器溢出时产生更新事件 UEV，当发生更新事件时，所有的影子寄存器都被立即更新，定时器状态寄存器 TIMx_SR 的更新中断标志位 UIF 被置为 1。

如图 7-13 所示，通用定时器的计数器由 CK_CNT 时钟驱动，当 TIMx_CR1 的 CEN 位置 1 时，计数器开始工作，CK_CNT 由预分频器给出，预分频器的时钟源 CK_PSC 可以选择多种时钟源提供。通用定时器中，TIM2、TIM3 和 TIM4 支持内部时钟(CK_INT)、外部输入脚触发(TI1 和 TI2)、内部定时器触发(ITR0～ITR3)和外部触发(ETR)四种。TIM9 支持上述四种，且其外部触发 ETR 在微控制器内部已被连接到 LSE 时钟；TIM10 和 TIM11 只支持 CK_INT、TI1 和 ETR，TIM10 和 TIM11 的 ETR 也连接到了 LSE 时钟；这样 TIM9、TIM10 和 TIM11 可以在微控制器休眠的状态下(CK_INT 被关闭)采用 LSE 时钟继续工作。

通常我们使用内部时钟 CK_INT 作为定时器的时钟源，如果从模式控制寄存器 TIMx_SMCR 的 SMS=000，只要 CEN 位被写成 1，预分频器的时钟就由内部时钟 CK_INT 提供。

当 TIMx_SMCR 寄存器的 SMS=111 时，外部触发模式 1 被选中，计数器可以在选定触发输入端的每个上升沿或下降沿计数。触发源共有 8 个，如图 7-14 所示，分别为 4 个其他定时器输出 ITRx、滤波后的定时器输入通道 TI1FP1、滤波后的定时器通道 TI2FP2、滤波后定时器边缘检测输入通道 TI1F_ED 和滤波后的外部触发源 ETRF，由 TIMx_SMCR 寄存器的 TS 域进行配置。如果选用 ITRx 作为时钟源，即选用了内部定时器触发作为时钟源，即定时器级联方式。在使用该模式前，需要先配置好外部触发源的信号。

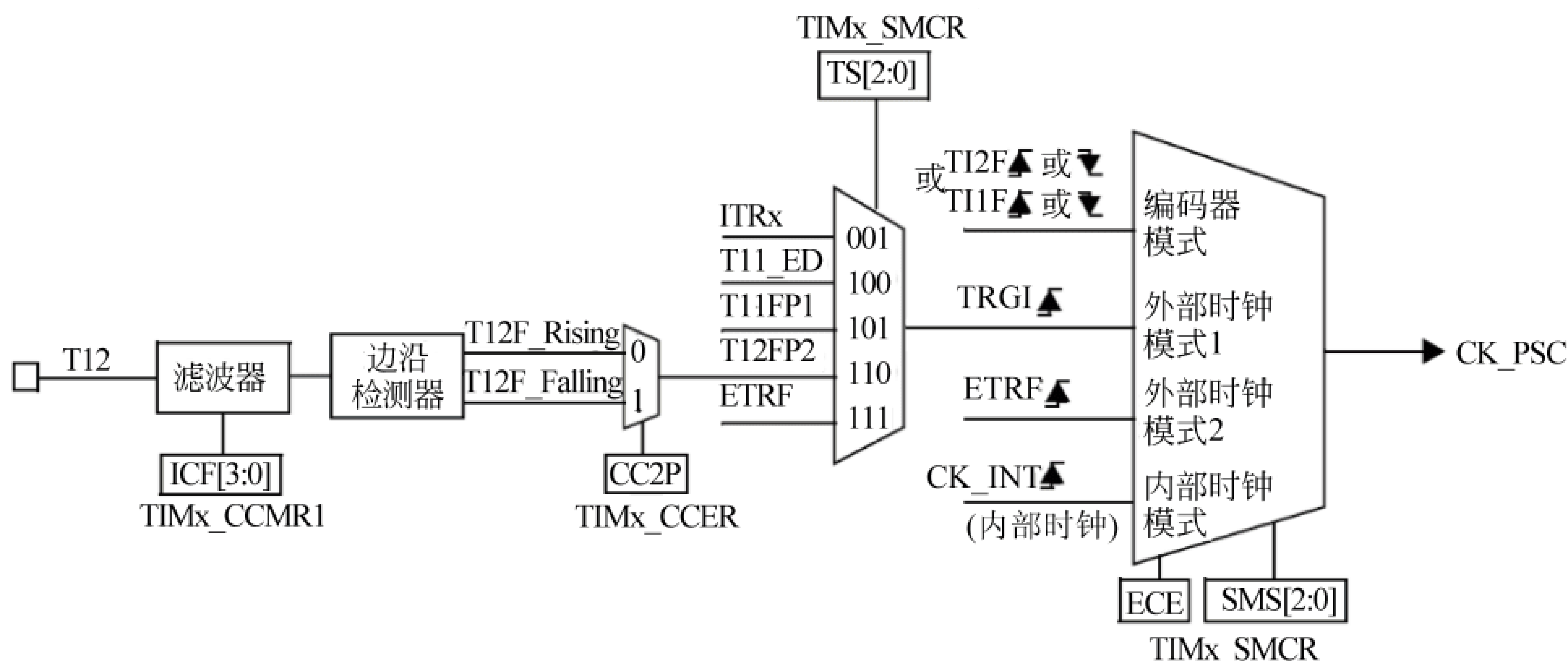


图 7-14　外部触发模式 1 的触发源

在外部触发模式 1 中，TIM10 和 TIM11 的外部时钟模式 1 触发源只能使用 TI1FP1、TI1F_ED 和 ETR。此外，不同于 TIM2、TIM3 和 TIM4 的时钟输入 TIx 只能由外部引脚输入，TIM9 的 TI1 输入时钟除了外部管输入脚 TIx 外(外部触发模式 1)，还支持 LSE(外部触发模式 2)；TIM10 的 TI1 输入时钟除了外部引脚输入外，还支持 LSE、LSI 和 RTC 唤醒中断；TIM11 的 TI1 输入时钟除了外部引脚输入外，还支持 MSI 和 HSE_RTC；这些输入时钟由寄存器 TIMx_OR 进行配置。

如图 7-14 所示，要选择 TI2FP2 上升沿作为定时器的时钟源，则需要配置 TIMx_CCMR1 寄存器的的 CC2S、IC2F 以及 TIMx_CCER 寄存器的输入极性 CC2P、CC2NP 等，

然后在 TIMx_SMCR 中配置触发源为 TI1FP2(TS=110),模式为外部触发模式 1(SMS=111),配置 TIMx_CR1 寄存器 CEN=1 启动定时器后,TI2FP2 驱动定时器工作,其工作时序如图 7-15 所示(TIF 为触发中断标志,当检测到一个 TI2 输入时,TIF 被置 1,通过软件进行清 0)。

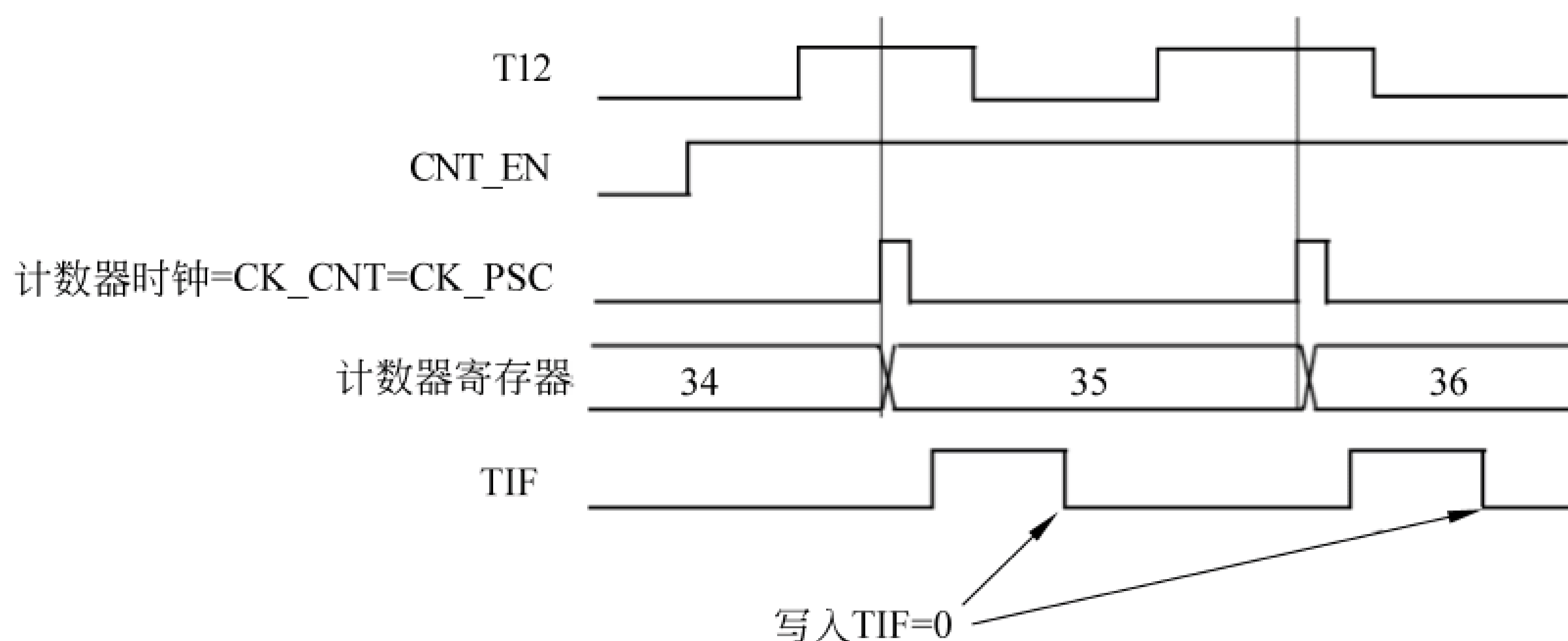


图 7-15　外部时钟模式下计数器时序

当 TIMx_SMCR 寄存器中的 ECE=1 时,外部触发模式 2 被选中,计数器在外部触发 ETR 的每一个上升沿或下降沿计数。如图 7-16 所示,外部时钟源 ETR 输入通路上带有分频器、滤波器,滤波器以一个可配置的基准频率对输入信号进行采样,当连续采样到 N 次有效电平时,认为一次有效的输入电平。

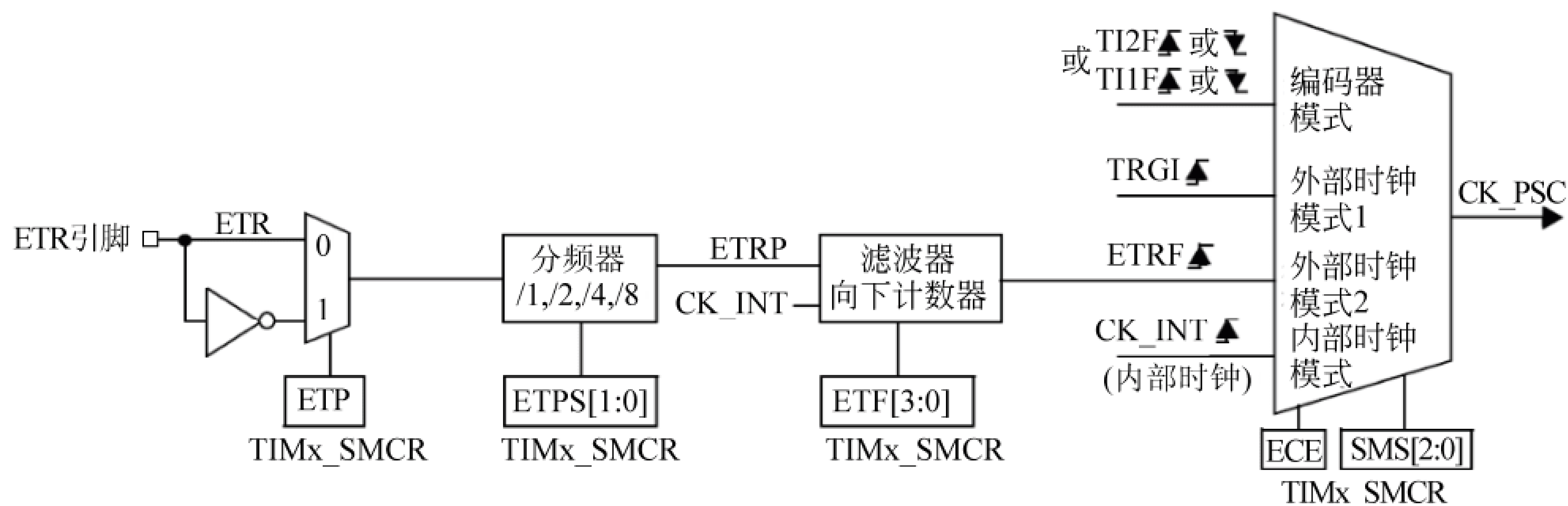


图 7-16　外部时钟源 ETR 触发电路

例如,配置外部 ETR 触发,2 个 ETR 上升沿进行一次计数,需要将 TIMx_SMCR 寄存器的 ETPS 置为 01,ETP 置为 0,配置 ECE=1 启用 ETR 外部时钟模式 2,启动定时器后,每两个 ETR 上升沿进行计数,时序如图 7-17 所示。

从图 7-14 可见,外部触发源 ETR 也可以在外部触发模式 1 下作为输入,此时和外部触发模式 2 下的功能一样。但如果需要使用 ETR 外部触发和从模式中的复位、触发、门控等进行组合应用,此时 SMS 已被配置为复位、触发或门控,无法配置 ETR 输入,此时可以配置 ECE 启用 ETR 作为时钟源。当外部模式 2 开启时,内部时钟和外部时钟模式 1 无效。

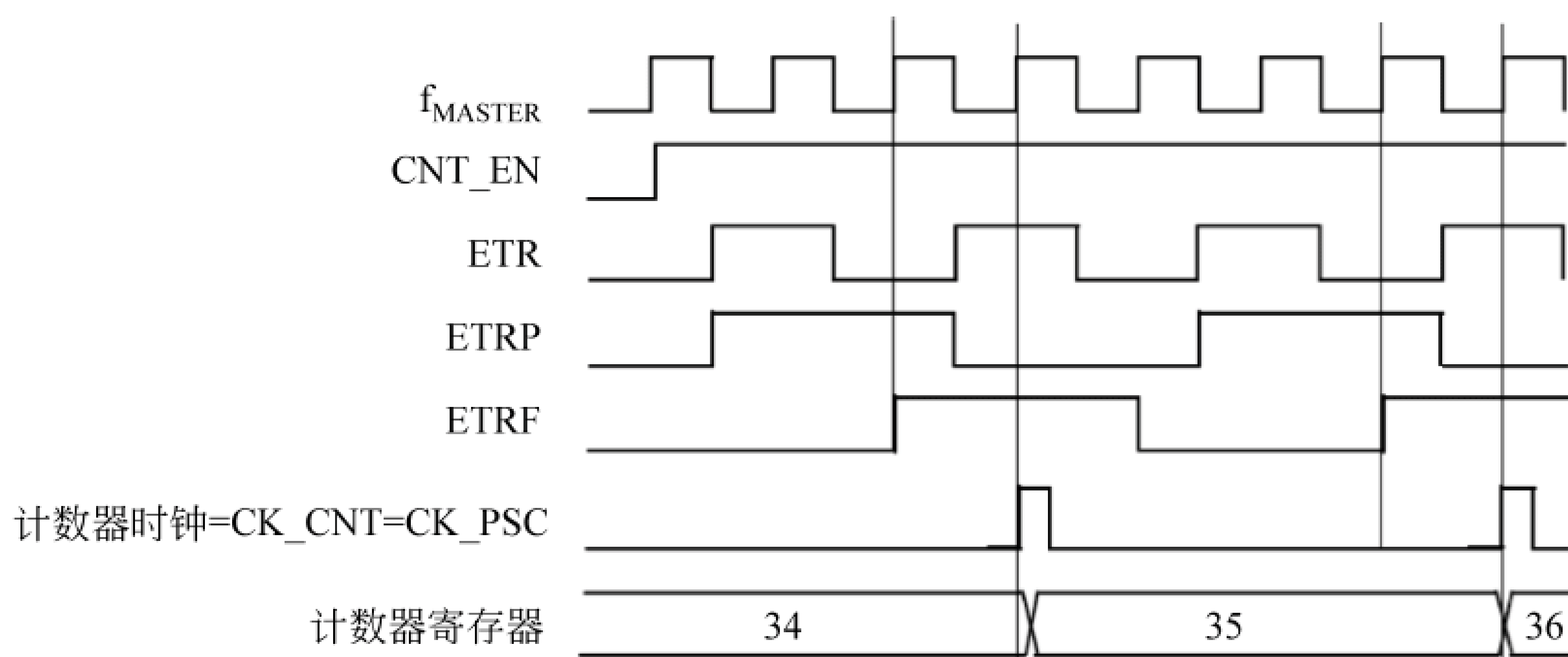


图 7-17　外部时钟触发模式 2 时序

7.5.2　通用定时器输入捕获和输出比较单元

输入捕获和输出比较是通用定时器较为复杂的功能,捕获可以用来对输入信号进行测量,输出比较可以用来产生特定的输出信号。每个定时器支持的捕获比较通道数量不同,但其组成结构类似。如图 7-9 所示,每个捕获通道的核心是一个捕获比较寄存器,它的输入部分主要是边沿检测电路,输出部分是输出控制电路。对于一个通道而言,只能选用输入捕获或输出比较中的一种模式。捕获/比较寄存器带有影子寄存器,读写过程仅操作其预装载寄存器。捕获模式下,捕获发生在影子寄存器上,然后再复制到预装载寄存器中;在比较模式下,预装载寄存器的内容被复制到影子寄存器中,影子寄存器和计数器进行比较判断。

1. 输入捕获

图 7-18 为捕获模式下输入电路的结构,输入部分对 TIx 引脚输入信号采样,产生一个滤波后的信号 TIxF,采样频率 fdts 由寄存器 TIMx_CR1 配置。然后,一个带极性选择的边缘检测器产生一个信号(TIxFPx)。通道选择器决定送到比较通道的信号源,该信号通过预分频后作为最终的输入信号 ICxPS 送到捕获/比较寄存器。滤波的目的是通过多次采样减少信号高频部分带来的抖动干扰。

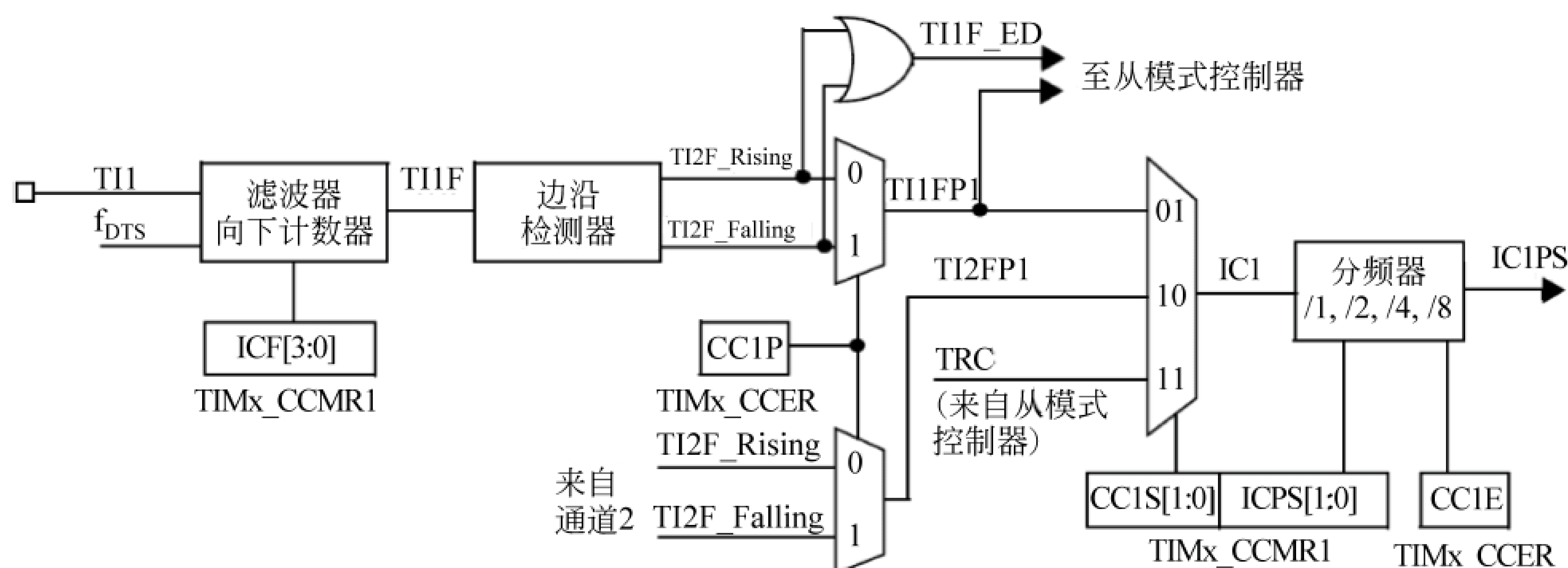


图 7-18　输入捕获电路结构

在输入捕获模式下，当检测到 ICx 信号上相应的边沿后，计数器的当前值被锁存到捕获/比较寄存器(TIMx_CCRx)中，同时产生捕获事件，置 TIMx_SR 寄存器相应的标志 CCxIF 为 1，如果使能了中断或 DMA，则将产生中断或者 DMA 操作。如果捕获事件发生时 CCxIF 标志已经为 1，则重复捕获标志 CCxOF 被置 1。读取存储在 TIMx_CCRx 寄存器中数据时 CCxIF 被清 0。

配置示例：在 TI1 输入端口输入一个频率为 1MHz 的方波，输入信号在最多 5 个内部时钟周期的时间内抖动，配置输入捕获寄存器在信号的上升沿捕获计数器的值并计算周期。

(1) 配置 TI1 输入端口的 GPIO 相关寄存器，将端口设置为复用输入模式。

(2) 选择输入端口：配置 TIMx_CCR1 寄存器的 CC1S=01，选择 TI1 作为捕获输入源。

(3) 配置输入滤波：配置滤波采样频率为 CK_INT，输入信号抖动在 5 个时钟周期内，因此 TIMx_CCMR1 寄存器中的 IC1F 置为 0011，即连续采样 8 次对输入信号上升沿进行判断。

(4) 选择转换边沿：在 TIMx_CCER 寄存器中写入 CC1P=0，捕获上升沿。

(5) 配置预分频器：如果希望 n(1,2,4,8)次信号上升沿捕获产生一个有效的电平转换时刻，配置预分频器 IC1PS=00,01,10 和 11。

(6) 开启捕获：设置 TIMx_CCER 寄存器的 CC1E=1，允许捕获。

(7) 开启中断：设置 TIMx_DIER 寄存器中的 CC1IE 和 CC1DE 位允许相关中断请求和 DMA 请求。

(8) 记录两次捕获的计数值，即可算出输入信号的频率。

2. 输出比较

如图 7-19 所示，输出比较电路的输出结果实际是由 OCxREF 决定的，比较寄存器和计数器的值相等时，输出一个 OCxREF 电平，电平由 OCxM 决定，可以是保持原有电平，设置为低电平，设置为高电平或者进行翻转，这个输出电平也可以被外部触发时钟源进行控制。OCxREF 的信号可以被极性选择器进行翻转，即低电平变高电平，高电平变低电平，而电平信号是否输出最终取决于输出使能电路，只有 TIMx_CCER 的 CCxE 域为 1 时 OC1 才真正输出。

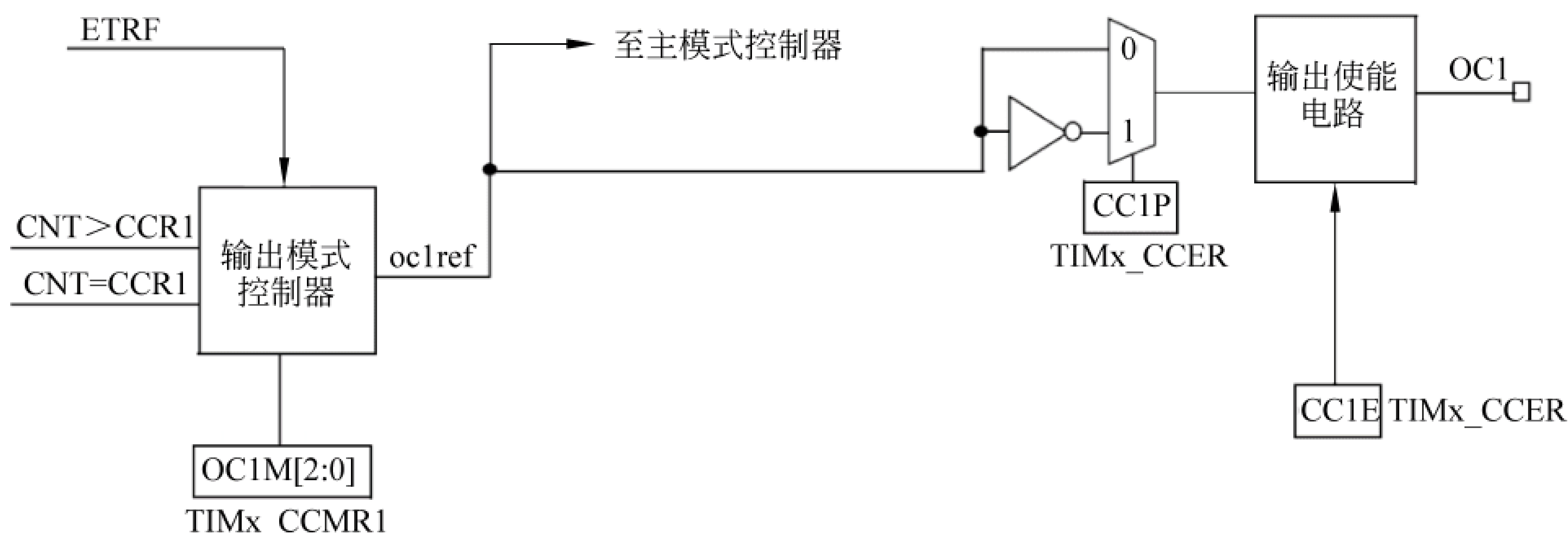


图 7-19 输出比较电路

当计数器与捕获/比较寄存器的内容相同时，输出比较功能做如下操作：

- 根据比较模式(TIMx_CCMRx 寄存器中的 OCxM 域)和输出极性(TIMx_CCER 寄

存器中的 CCxP 位)的定义,将 OCxREF 的值输出到对应的引脚上。

• 设置中断状态寄存器中的标志位(TIMx_SR 寄存器中的 CCxIF 位)。
• 若设置了中断屏蔽(TIMx_DIER 寄存器中的 CCxIE 位)允许中断,则产生一个中断请求。

在输出比较模式下,只有比较事件才会影响输出结果,更新事件对 OCxREF 和 OCx 输出没有影响。

输出比较模式的配置步骤一般为:

(1) 选择计数器时钟(内部、外部,设置预分频器);
(2) 将相应的数据写入 TIMx_ARR 和 TIMx_CCRx 寄存器中;
(3) 设置 CCxIE、CCXDE 是否产生中断请求/或 DMA 请求;
(4) 配置输出模式 OCxM;
(5) 设置 TIMx_CR1 寄存器的 CEN 位启动计数器。

3. PWM 输出

PWM 指脉冲宽度调制,也就是占空比可变的脉冲波形,PWM 被广泛应用在电机控制、频率调节等领域。PWM 模式下,可以产生一个由 TIMx_ARR 寄存器确定频率、由 TIMx_CCRx 寄存器确定占空比的信号。例如,当 CNT 的值小于 TIMx_CCRx 中的值时候输出高电平或低电平,当大于的时候反向,如图 7-20 所示。

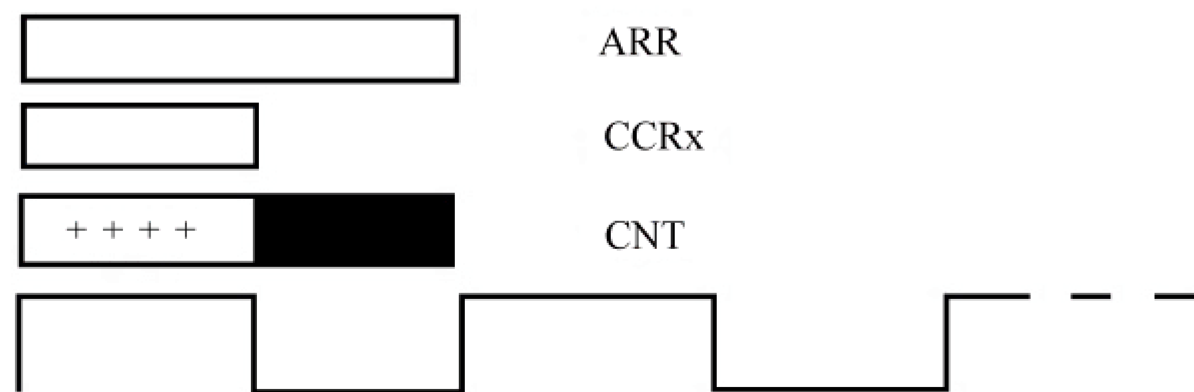


图 7-20　PWM 示意图

TIMx_ARR 寄存器设定脉冲周期,TIMx_CCR 设定占空比,CNT 计数器最大值为 TIMx_ARR,由于 TIM 有多于 1 个的捕获比较通道,因此一个定时器可以同时输出多个同一频率但占空比不同的 PWM 波形。

TIMx_CCMRx 寄存器中的 OCxM 支持两种 PWM 模式:

(1) PWM 模式 1:在向上计数时,一旦 CNT＜CCRx,通道 x 为高电平(OC1REF＝1),否则为低电平(OC1REF＝0);在向下计数时,一旦 CNT＞CCRx,通道 x 为低电平,否则为高电平(OC1REF＝1)。

(2) PWM 模式 2:在向上计数时,一旦 CNT＜CCRx,通道 x 为低电平,否则为高电平;在向下计数时,一旦 CNT＞CCRx,通道 x 为高电平,否则为低电平。

在 PWM 模式下,必须设置 TIMx_CCMRx 寄存器 OCxPE 位为使能预装载寄存器,设置 TIMx_CR1 寄存器的 ARPE 位使能自动重装载的预装载寄存器,因此启用计数器之前须通过设置 TIMx_EGR 寄存器中的 UG 位来手动初始化所有的寄存器。

图 7-21 为 ARR 配置为 8 时,PWM 模式 1 的示例。当 CNT＜CCRx 时,PWM 输出

OCxREF 为高,否则为低。如果 CCRx 的值大于自动重装载值(TIMx_ARR),则输出一个占空比 100%的信号,即 OCxREF 始终为 1。如果 CCRx=0,则产生占空比 0%的信号,即 OCxREF 始终保持为 0。

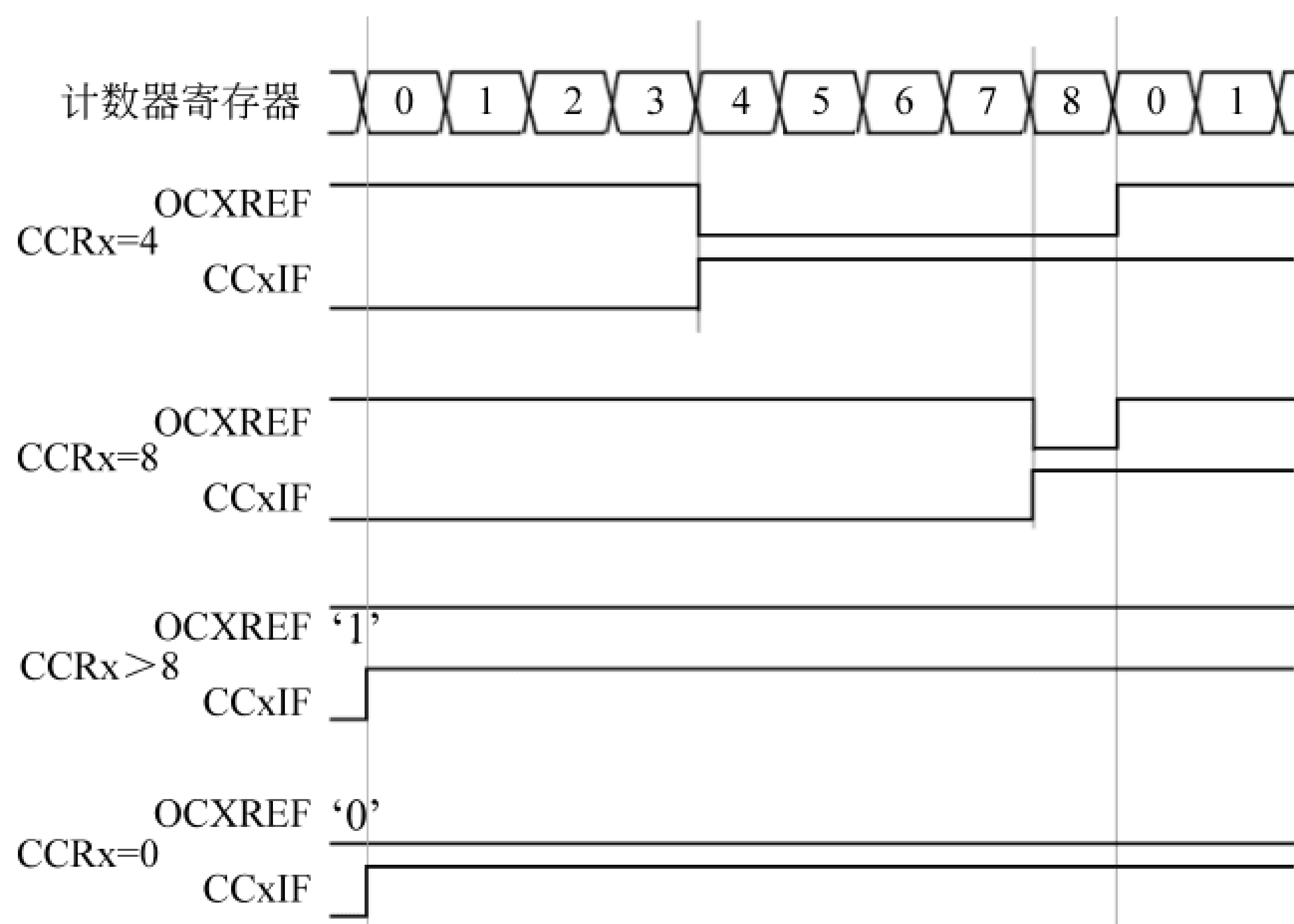


图 7-21　PWM 模式 1 时序

4. PWM 输入模式

PWM 输入模式用来测量 PWM 的周期和占空比,该模式是输入捕获模式的一个特例,其与不同输入捕获的区别在于:

- 一个 TIx 的输入被连接到两个不同的 ICx 通道上;
- 两个 ICx 输入信号配置为边沿有效,但极性相反;
- 一个 TIxFP 信号被作为触发输入信号,从模式控制器被配置成复位模式。

由于只有 TI1FP1 和 TI2FP2 连到了从模式控制器,所以 PWM 输入模式只能使用 TIMx_CH1 /TIMx_CH2 信号。将输入 PWM 信号连接到 TI1,则信号周期存储在 TIMx_CCR1 寄存器,占空比存储在 TIMx_CCR2 寄存器,具体配置步骤如下:

(1) 选择 TIMx_CCR1 的有效输入,TIMx_CCMR1 的 CC1S=01(选择 TI1);

(2) 选择 TI1FP1 的极性为上升沿有效,置 CC1P=0;

(3) 选择 TIMx_CCR2 的有效输入,TIMx_CCMR1 的 CC2S=10(选择 TI1);

(4) 选择 TI1FP2 的有效极性为下降沿有效,置 CC2P=1;

(5) 选择外部触发时钟为 TIFP1,置 TIMx_SMCR 的 TS=101;

(6) 配置从模式控制器为复位模式,置 TIMx_SMCR 中的 SMS=100;

(7) 使能捕获,置 TIMx_CCER 的 CC1E=1 且 CC2E=1。

如图 7-22 所示,当 T1 检测到上升沿时,复位定时器,CNT 从 0 开始计数,当下降沿到达时,IC2 捕获到此时的计数值,即为 PWM 高电平宽度,当下一个上升沿到达时,IC1 捕获到此时的计数值,即为 PWM 的周期,通过定时器时钟频率和捕获的计数值即可算出 PWM 信号的实际周期和占空比。

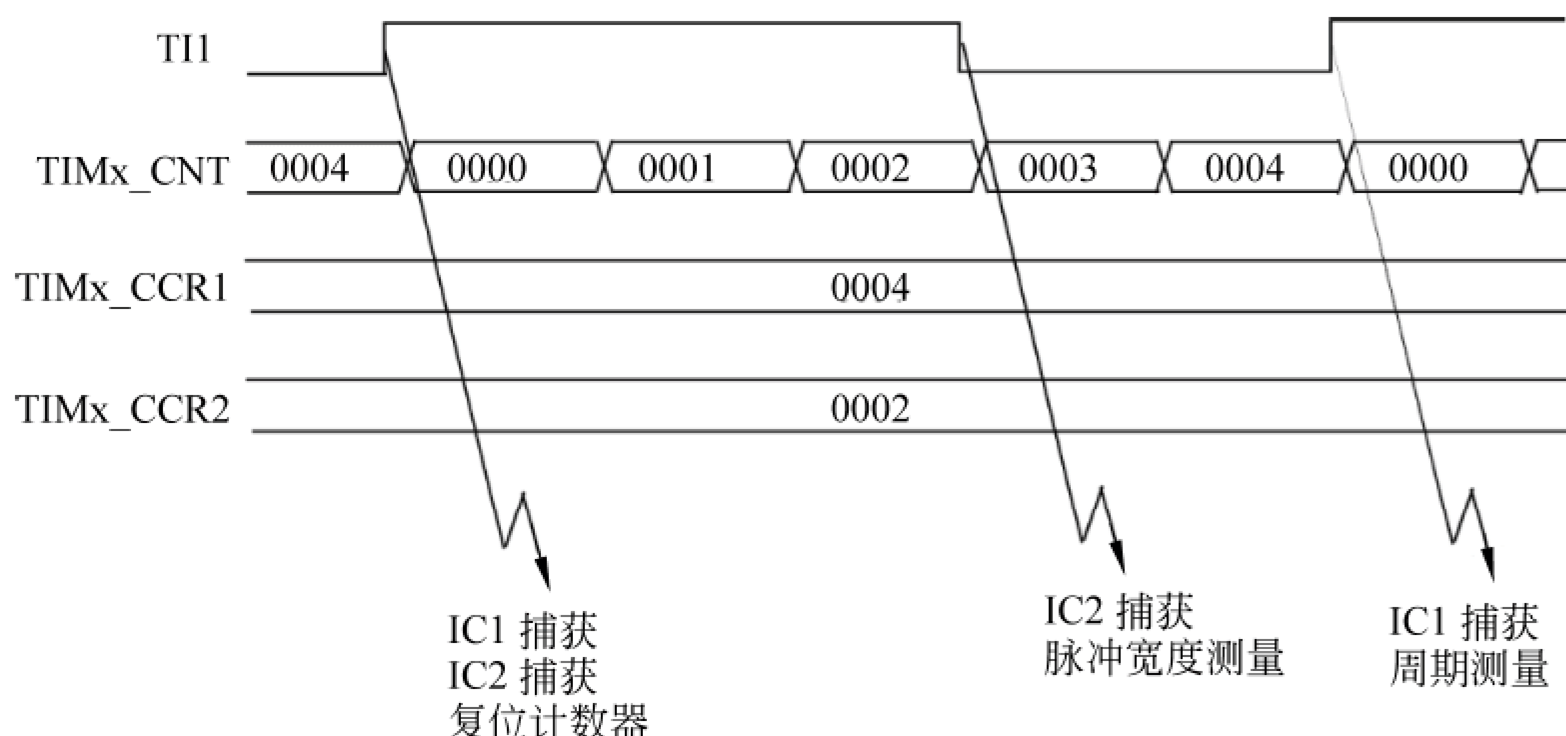


图 7-22　PWM 输入测量

7.5.3　TIMx 的外部触发同步模式

TIMx 在从模式可以通过外部触发信号同步，从模式包括复位、门控和触发模式。

1. 复位模式

外部触发输入事件发生时，计数器和它的预分频器被重新被初始化，如果 IMx_CR1 寄存器的 URS 位为 0，即从模式控制器可以产生更新事件，此时还产生一个更新事件 UEV；然后所有的预装载寄存器(TIMx_ARR，TIMx_CCRx)都被更新。

如图 7-23 所示，TI1 上升沿作为触发，产生 UEV 事件，计数器被清 0，TIF 标志表示产生了一个触发中断，如果中断允许，则向 MCU 发起中断请求。

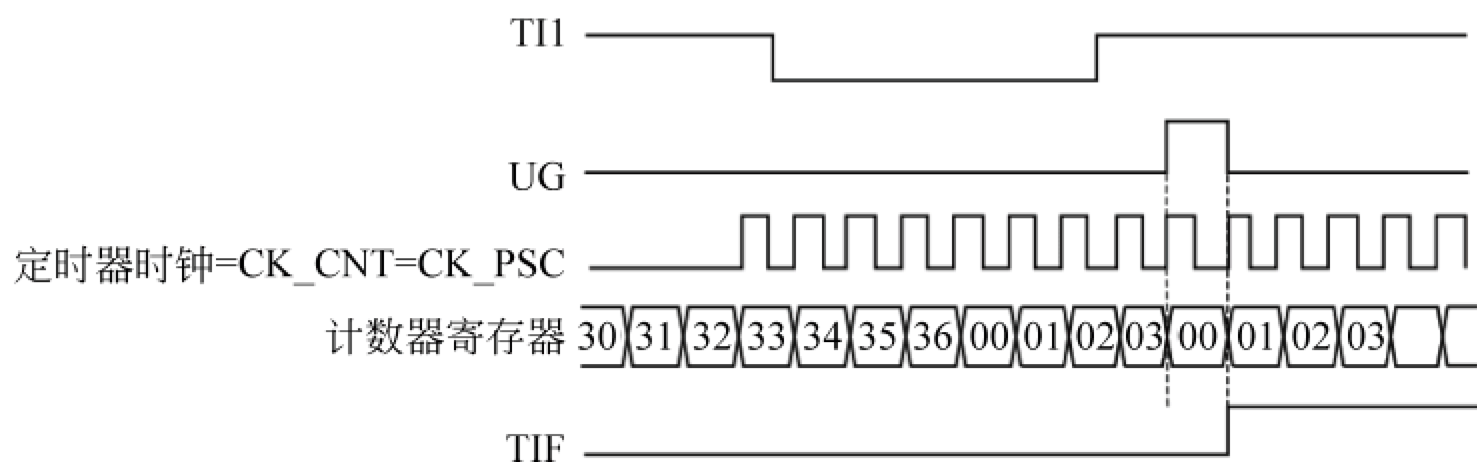


图 7-23　复位模式的控制时序

2. 门控模式

计数器的使能依赖于选中的输入端的电平。如下图 7-24 所示，TI1 的低电平作为触发源，配置通道 1 作为 TI1 低电平检测，当 TI1 为高电平时，计数器停止计数，当检测到 TI1 变为低电平后，计数器立即开始工作。在计数器停止和启动均会引起 TIF 标志置 1，表示产生一个触发中断。

3. 触发模式

计数器的使能依赖于选中的输入端上的事件。如图 7-25 所示，配置 TI2 的上升沿作为外部触发源，当检测到上升沿时，计数器启动，开始计数，并设置触发中断标志 TIF，否则定

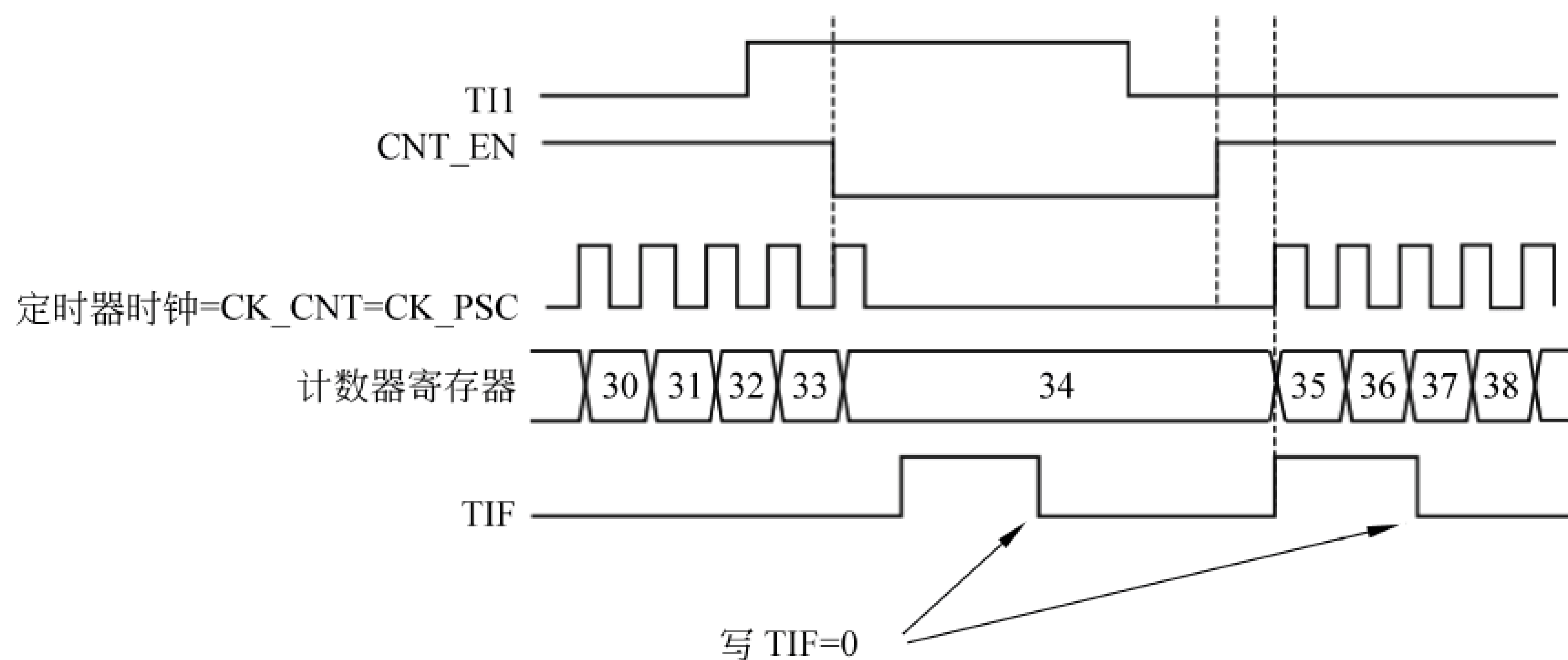


图 7-24　门控模式的控制时序

时器停止工作。

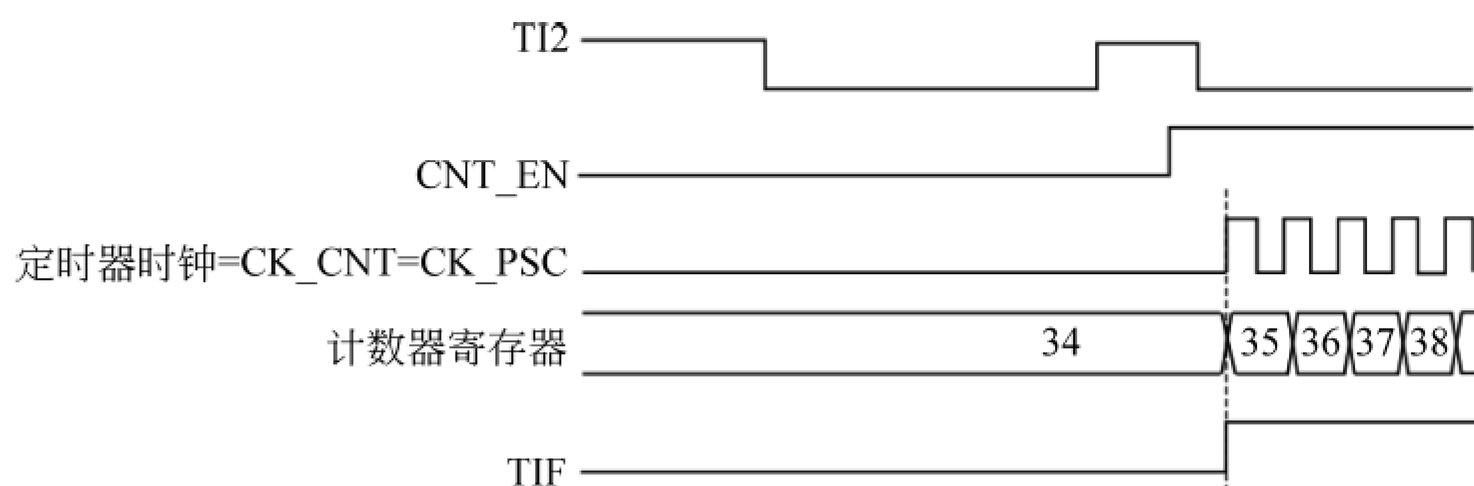


图 7-25　触发模式的控制时序

4. 外部时钟 2+触发模式

外部时钟模式 2 可以与另一种从模式(外部时钟模式 1 和编码器模式除外)一起使用。这时,ETR 信号被用作外部时钟的输入,在复位模式、门控模式或触发模式可以选择另一个输入作为触发输入。如图 7-26 所示,TI1 上升沿作为触发信号,ETR 作为定时器时钟,当检测到 TI1 的上升沿后,定时器启动,在 ETR 的控制下开始计数。

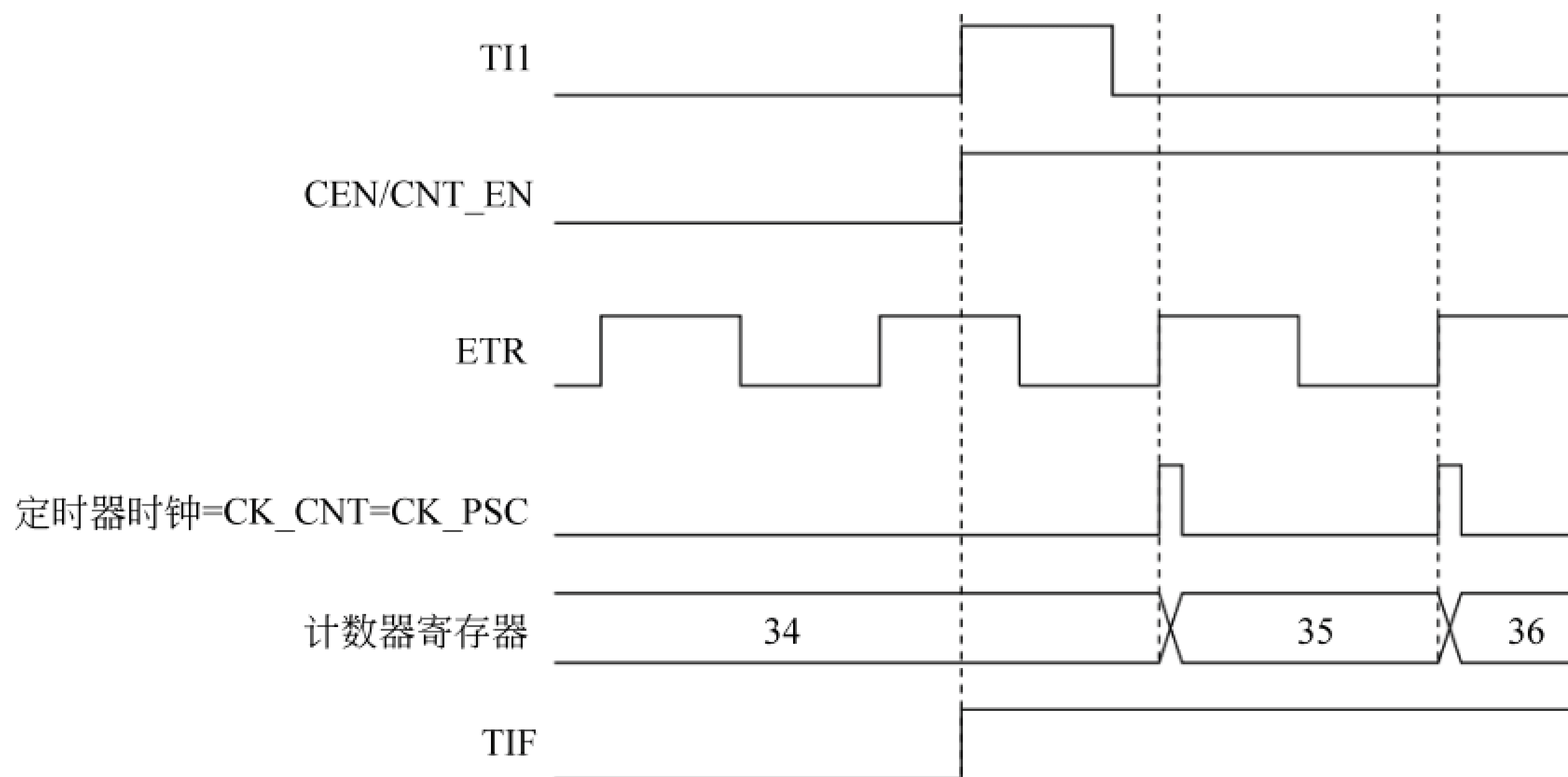


图 7-26　外部时钟模式 2+触发模式的控制时序

7.6　定时器寄存器

通用定时器包括 16 个寄存器，TIM2～TIM4，TIM6～TIM7 以及 TIM9～TIM11 各自对每个寄存器的域的支持有所不同，TIM9～TIM11 还有自己特殊的寄存器，下面对常用寄存器进行介绍，其中不同定时器对不同域的支持在寄存器域说明中进行标注。定时器常用寄存器如表 7-4 所示。

表 7-4　定时器寄存器及其功能

寄存器名	读写权限	功　能
TIMx_CR1	RW	定时器控制寄存器 1
TIMx_CR2	RW	定时器控制寄存器 2
TIMx_SMRC	RW	定时器从模式选择寄存器
TIMx_DIER	RW	定时器 DMA 和中断使能寄存器
TIMx_SR	RW	定时器状态寄存器
TIMx_EGR	W	定时器事件产生寄存器
TIMx_CCMR1	RW	捕获/输出模式寄存器 1
TIMx_CCMR2	RW	捕获/输出模式寄存器 2
TIMx_CCER	RW	捕获/输出使能寄存器
TIMx_CNT	RW	计数器
TIMx_PSC	RW	预分频寄存器
TIMx_ARR	RW	自动重装载寄存器
TIMx_CCR1	RW	通道 1 捕获/比较寄存器
TIMx_CCR2	RW	通道 2 捕获/比较寄存器
TIMx_CCR3	RW	通道 3 捕获/比较寄存器
TIMx_CCR4	RW	通道 4 捕获/比较寄存器

1. 控制寄存器 TIMx_CR1

控制寄存器 TIMx_CR1 是 16 位寄存器，如图 7-27 所示，其有效域包括：

CKD[1：0]：时钟分频因子，用于配置 ETR、TIx 的数字滤波器采样频率 fdts 和定时器

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						CKD[1:0]		ARPE	CMS		DIR	OPM	URS	UDIS	CEN
						rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 7-27　控制寄存器 1

内部时钟(CK_INT)频率之间的分频比例。00 表示不分频,01 表示二分频,10 表示 4 分频。

ARPE:自动重装载预装载允许位,0 表示 TIMx_ARR 寄存器没有缓冲,写入 ARR 寄存器将直接改变影子寄存器的值,1 表示 TIMx_ARR 寄存器被装入缓冲器,只有事件发生时 ARR 寄存器值才会被加载到影子寄存器。

CMS[1:0]:选择中央对齐的四种模式,分别为不使用中央对齐,中央对齐模式 1、中央对齐模式 2 和中央对齐模式 3,具体见 STM32L1xx 参考手册文档。

DIR:计数方向,0 表示向上计数,1 表示向下计数,当选用中央对齐模式时,该位为只读,表示计数方向。当配置向上计数时,配置 CMS=00,DIR=0。

OPM:单脉冲模式,0 表示在发生更新事件时,计数器不停止,1 表示在下一次更新事件发生时清除 CEN 位,计数器停止。

URS:更新请求源,设置产生中断的事件源,0 表示计数器溢出、设置 UG 位以及从模式控制器产生的更新都可以产生中断,1 表示只有计数器溢出才产生更新中断。

UDIS:禁止更新,0 表示允许产生更新事件,1 表示不产生更新事件。

CEN:计数器启动位, 0 表示禁止计数器,1 表示启动计数器。

其中,基本定时器 TIM6 和 TIM7 没有 CMS 和 DIR 域。

2. 控制寄存器 TIMx_CR2

控制寄存器 TIMx_CR2 是 16 位寄存器,如图 7-28 所示,其有效域包括:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								TI1S	MMS[2:0]			CCDS	Reserved		
								rw	rw	rw	rw	rw			

图 7-28 控制寄存器 2

TI1S:TI1 输入源选择,0 表示 TIMx_CH1 引脚连到 TI1 输入,1 表示 TIMx_CH1、TIMx_CH2 和 TIMx_CH3 引脚经异或后连到 TI1 输入。

MMS[2:0]:主模式选择,用于选择在主模式下送到从定时器的同步信息 TRGO 的来源。

- 000 表示复位,TIMx_EGR 寄存器的 UG 位被用于作为触发输出(TRGO)。
- 001 表示使能,计数器使能信号 CNT_EN 被用于作为触发输出(TRGO)。
- 010 表示更新,更新事件 UE 被选为触发输入(TRGO)。
- 011 表示比较脉冲,在发生一次捕获或一次比较成功时,当要设置 CC1IF 标志时,触发输出送出一个正脉冲(TRGO)。
- 100~111 表示 OC1REF~OC4REF 信号被用于作为触发输出(TRGO)。

其中,基本定时器 TIM6 和 TIM7 只有 MMS 域。TIM10、TIM11 没有此寄存器。

3. 从模式控制寄存器(TIMx_SMCR)

从模式控制寄存器 TIMx_SMCR 为 16 位寄存器,如图 7-29 所示,其主要域包括:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ETP	ECE	ETPS[1:0]		ETF[3:0]				MSM	TS[2:0]			OCCS	SMS[2:0]		
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 7-29 从模式控制寄存器

ETP：外部触发极性，用于选择是用ETR还是ETR的反相来作为触发操作，0表示ETR不反相，高电平或上升沿有效，1表示1，ETR反相，低电平或下降沿有效。

ECE：外部时钟使能位，1表示启用外部时钟模式2，计数器由ETRF信号上的任意有效边沿驱动，0表示禁用外部时钟模式2。

ETPS[1：0]：外部触发预分频系数，外部触发信号ETRP的频率最高不能超过CK_INT/4。ETRP过快时，使用预分频降低ETRP的频率。00表示关闭预分频，01～11分别表示2、4、8分频。

ETF[3：0]：外部触发滤波，定义对ETRP信号采样的频率和对ETRP数字滤波的带宽。数字滤波带宽用N表示，即N个事件后会产生一个输出的跳变，N的取值为2、4、5、6、8。采样频率可以选择fCK_INT或fDTS的2、4、8、16分频，其中fDTS由TIMx_CR1的CDK配置。具体见STM32L1xx参考手册文档。

TS[2：0]：外部模式1的触发时钟源选择，000-011表示内部定时器TIM1～TIM4作为时钟源，100～110为输入通道1的不同信号作为时钟源，111表示外部触发输入ETRF。TS和SMS配合使用，SMS=000时TS无效。

SMS[2：0]：从模式选择，000表示不使用从模式，预分频器直接由内部时钟驱动，100～110为复位模式、门控模式和触发模式，111表示使用外部时钟模式1，具体外部时钟源由TS选择。

基本定时器TIM6、TIM7没有这个寄存器，通用定时器TIM9没有OCCS域。TIM10、TIM11只有8～15位，没有MSM、TS、OCCS以及SMS域。

4. DMA/中断使能寄存器TIMx_DIER

中断使能寄存器为16位寄存器，如图7-30所示，其有效域为：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	TDE	Res	CC4DE	CC3DE	CC2DE	CC1DE	UDE	Res.	TIE	Res	CC4IE	CC3IE	CC2IE	CC1IE	UIE
	rw		rw	rw	rw	rw	rw		rw		rw	rw	rw	rw	rw

图7-30 中断使能寄存器

TDE：允许触发DMA请求，0表示禁止，1表示允许。

CC4DE～CC1DE：允许捕获/比较通道x的DMA请求，0表示禁止，1表示允许。

UDE：允许更新的DMA请求，0表示禁止，1表示允许。

TIE：触发中断使能，0表示禁止，1表示允许。

CC4IE～CC1IE：允许捕获/比较通道x中断，0表示禁止，1表示允许。

UIE：允许更新中断，0表示禁止，1表示允许。

其中，基本定时器TIM6和TIM7只有UDE和UIE两个域。TIM9只有TIE、CC2IE、CC1IE和UIE域，TIM10和TIM11只有CC1IE和UIE域。

5. 状态寄存器TIMx_SR

状态寄存器TIMx_SR为16位寄存器，如图7-31所示，其有效域包括：

CC4OF～CC1OF：捕获/比较通道重复捕获标记，0表示无重复捕获，1表示重复捕获，

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			CC4OF	CC3OF	CC2OF	CC1OF	Reserved		TIF	Res	CC4IF	CC3IF	CC2IF	CC1IF	UIF
			rc_w0	rc_w0	rc_w0	rc_w0			rc_w0		rc_w0	rc_w0	rc_w0	rc_w0	rc_w0

图 7-31 状态寄存器

在输入捕获下,若计数器的值被捕获到 TIMx_CCR1 寄存器,但 CC1xF 的状态已经为 1,则 CCxOF 标记由硬件置 1,写 0 可清除该位。

TIF:触发器中断标记,当发生触发事件时由硬件置 1,软件写 0 清除。

CC4IF～CC1IF:捕获/比较通道中断标记,当通道 CCx 配置为输出模式,计数器值与比较值匹配时该位由硬件置 1,软件写 0 清除;当通道 CCx 配置为输入模式,当捕获事件发生时该位由硬件置 1,可由软件清 0 或通过读 TIMx_CCRx 清 0。

UIF:更新中断标记,当产生更新事件时该位由硬件置 1,它由软件清 0。

其中,基本定时器 TIM6、TIM7 只有 UIF 域,TIM9 只有 CC2OF、CC1OF、TIF、CC2IF、CC1IF 以及 UIF 域,TIM10 和 TIM11 只有 CC1OF、CC1IF 和 UIF 域。

6. 事件产生寄存器(TIMx_EGR)

事件产生寄存器用于软件触发事件,如图 7-32 所示,其有效域包括:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved									TG	Res.	CC4G	CC3G	CC2G	CC1G	UG
									w		w	w	w	w	w

图 7-32 事件产生寄存器

TG:产生触发事件,该位写 1 产生一个触发事件,由硬件自动清 0。

CC4G～CC1G:产生捕获/比较通道事件,该位写 1 产生一个捕获/比较事件,由硬件自动清 0。

UG:产生更新事件,该位写 1 产生更新事件,重新初始化计数器,预分频器计数器也被清 0 但预分频系数不变。

其中,基本定时器 TIM6 和 TIM7 只有 UG 域,TIM10 和 TIM11 只有 CC1G 和 UG 域,TIM9 只有 TG、CC2G、CC1G 和 UG 域。

7. 捕获/比较模式寄存器 TIMx_CCMR1

捕获比较寄存器 TIMx_CCMR1 用于配置捕获/比较通道 1 和 2,在配置成输入捕获和比较输出时使用同一个寄存器,但其域有不同的定义,其中 CCxS 域在两种模式下相同,CCxS 用于定义通道的方向。如图 7-33 所示,配置成输入捕获时,使用第二行的域定义,比较输出时,采用第一行的域定义。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OC2CE	OC2M[2:0]			OC2PE	OC2FE	CC2S[1:0]		OC1CE	OC1M[2:0]			OC1PE	OC1FE	CC1S[1:0]	
IC2F[3:0]				IC2PSC[1:0]				IC1F[3:0]				IC1PSC[1:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 7-33 捕获/比较模式寄存器

TIM6 和 TIM7 没有此寄存器，TIM10 和 TIM11 没有高 8 位。

CC1S[1：0]：捕获/比较 1 选择，定义通道的方向（输入输出）以及输入脚的选择，配置为 00 时通道为输出，01 表示输入，IC1＝TI1，10 表示输入，IC1＝TI2，11 时通道为输入，输入源有 TIMx_SMCR 的 TS 确定。CCxS 仅在通道关闭时（TIMx_CCER 寄存器的 CC1E＝0）才可以配置。

CC2S[1：0]：捕获/比较 2 选择，配置为 01 表示输入，IC2＝TI2，10 表示输入，IC2＝TI1，其余与 CC1S 相同。

输出比较模式下各个域的定义如下：

OCxCE：输出比较通道 x 清 0 使能，1 表示一旦检测到 ETRF 输入高电平，OCxREF 置为 0。

OCxM[2：0]：输出比较通道 x 的 OCxREF 模式选择，OCxREF 决定了 OCx 的输出值，而 OCx 的实际输出电平取决于 TIMX_CCER 寄存器的 CCxP 极性设置是否对 OC1REF 进行翻转。该域配置为 000 表示冻结，计数器与比较寄存器匹配不影响 OCxREF；001 表示计数器与比较寄存器匹配时，OC1REF 置为为高电平；010 表示匹配时置为低电平；011 表示匹配时翻转 OCxREF 的电平；100 和 101 分别表示强制输出为低电平和高电平；110 和 111 表示 PWM 的模式 1 和模式 2。

OCxPE：输出比较通道 x 预装载使能，0 表示禁止 TIMx_CCRx 寄存器的预装载功能，写入 TIMx_CCRx 寄存器的数值立即生效；1 表示开启 TIMx_CCRx 寄存器的预装载功能，TIMx_CCRx 的预装载值在更新事件到来时被传送至当前寄存器中。

OCxFE：输出比较通道 x 快速使能，仅在 PWM1 和 PWM2 模式使用。

输入捕获模式下寄存器的域定义如下：

ICxF[3：0]：输入捕获通道 x 的滤波器设置，定义了 TI1 输入的采样频率及数字滤波器长度。数字滤波器由一个事件计数器组成，记录到 *N* 个事件后会产生一个输出的跳变，*N* 的取值为 2，4，5，6，8，采样频率可以选择 fCK_INT 或 fDTS 的 2、4、8、16 分频，其中 fDTS 由 TIMx_CR1 的 CDK 配置，具体参考 STM32L1xx 参考手册。

ICxPSC[1：0]：输入/捕获通道 x 的预分频器，当 TIMx_CCER 寄存器的 CCxE 被清 0，预分频器复位。00 表示无预分频 01～11 分别表示 2、4、8 个事件触发一次捕获。

8．捕获/比较模式寄存器 TIMx_CCMR2

TIMx_CCMR2 与 TIMx_CCMR1 的域定义类似，区别在于 CC3S 配置为 01 时表示输入 TI3 连接到 IC3，10 表示 TI4 连接到 IC3；而 CC4S 配置为 01 时表示 TI4 被连接到 IC4，10 表示 TI3 被连接到 IC4。

9．捕获/比较使能寄存器 TIMx_CCER

如图图 7-34 所示，TIMx_CCER 的有效域定义如下：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CC4NP	Res.	CC4P	CC4E	CC3NP	Res.	CC3P	CC3E	CC2NP	Res.	CC2P	CC2E	CC1NP	Res.	CC1P	CC1E
rw		rw	rw	rw		rw	rw	rw		rw	rw	rw		rw	rw

图 7-34　捕获/比较使能寄存器

CC4P～CC1P：捕获/比较通道 x 的输出极性，通道 x 配置为输出时，该位配置为 0 表示 OCx 高电平有效，1 表示低电平有效；通道 x 配置为输入时，该位表示是选择 ICx 还是 ICx 的反相信号作为触发或捕获信号，0 表示不反相，1 表示反相。

CC4NP～CC1NP：捕获/比较通道 x 的输出极性，配置为输出时，该位无效，保持为 0；配置为输入时，与 CCxP 组合表示 TI1FP1 和 TI2FP1 的极性和触发电平：00 表示 TIxFP1 的上升沿，不反向，01 表示 TIxFP1 的下降沿，反向，11 表示双沿，反向，10 保留。

CC4E～CC1E：输入捕获通道 x 的输出使能，通道配置为输出时，该位配置为 0 表示禁止 OCx 输出；通道 x 配置为输入时，该位配置为 0 表示禁止捕获。

基本定时器 TIM6 和 TIM7 没有此寄存器，TIM9 只使用低 8 位，TIM10 和 TIM11 只有 CC1NP、CC1P 和 CC1E 域。

10. 计数器 TIMx_CNT

TIMx_CNT 寄存器如图 7-35 所示，16 位的 CNT 表示计数器值。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CNT[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 7-35　计数器寄存器

11. 预分频器 TIMx_PSC

TIMx_PSC 用于预分频控制，其寄存器如图 7-36 所示，16 位的 PSC 表示预分频值，计数器的时钟频率 CK_CNT 等于 fCK_PSC/(PSC[15：0]+1)。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PSC[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 7-36　预分频器寄存器结构

12. 自动重装载寄存器 TIMx_ARR

TIMx_ARR 寄存器如图 7-37 所示，有效域 16 位，表示 ARR 自动重装载值，当自动重装载的值为空时，计数器不工作。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ARR[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 7-37　自动重装载寄存器

13. 捕获/比较寄存器 TIMx_CCR1、TIMx_CCR2、TIMx_CCR3、TIMx_CCR4

捕获/比较寄存器 TIMx_CCRx 如图 7-38 所示，有效域 CCRx 表示捕获/比较 1 的值，若 CC1 通道配置为输出，CCR1 包含了装入当前捕获/比较 1 寄存器的值(预装载值)，如果在 TIMx_CCMR1 寄存器 OCxPE=0，写入改寄存器的值会被立即传输至当前寄存器中，否则只有当更新事件发生时，才传输至当前当前寄存器中。当前捕获/比较寄存器与计数器 TIMx_CNT 比较，并在 OC1 端口上产生输出信号。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CCR1[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 7-38　捕获/比较寄存器 1

若 CCx 通道配置为输入，CCRx 为上一次输入捕获事件(ICx)记录的计数器值。

TIM6 和 TIM7 没有此寄存器，TIM9 有 CCR1 和 CCR2 两个寄存器，TIM10、TIM11 只有 CCR1 一个寄存器。

7.7　外围定时器库函数

为了便于对通用和基本定时器进行操作，CMSIS 提供了定时器的寄存器定义和库函数。

定时器的寄存器结构体定义 stm32l1xx.h 头文件中，我们可以通过结构体类型 TIM_TypeDef 定义变量 TIMx 对定时器进行控制。

```
typedef struct
{
  uint16_t  CR1;                        //控制寄存器 1
  uint16_t  RESERVED0;
  uint16_t  CR2;                        //控制寄存器 2
  uint16_t  RESERVED1;
  uint16_t  SMCR;                       //从模式控制寄存器
  uint16_t  RESERVED2;
  uint16_t  DIER;                       //中断和 DMA 使能寄存器
  uint16_t  RESERVED3;
  uint16_t  SR;                         //状态寄存器
  uint16_t  RESERVED4;
  uint16_t  EGR;                        //事件产生寄存器
  uint16_t  RESERVED5;
  uint16_t  CCMR1;                      //捕获/比较模式寄存器 1
  uint16_t  RESERVED6;
  uint16_t  CCMR2;                      //捕获/比较模式寄存器 2
  uint16_t  RESERVED7;
  uint16_t  CCER;                       //捕获/比较使能寄存器
  uint16_t  RESERVED8;
  uint32_t  CNT;                        //计数寄存器
  uint16_t  PSC;                        //预分频寄存器
  uint16_t  RESERVED10;
  uint32_t  ARR;                        //自动重装载寄存器
  uint32_t  RESERVED12;
  uint32_t  CCR1;                       //捕获/比较寄存器 1
```

```
    uint32_t  CCR2;                    //捕获/比较寄存器 2
    uint32_t  CCR3;                    //捕获/比较寄存器 3
    uint32_t  CCR4;                    //捕获/比较寄存器 4
    uint32_t  RESERVED17;
    uint16_t  DCR;                     //DMA 控制寄存器
    uint16_t  RESERVED18;
    uint16_t  DMAR;                    //DMA 地址寄存器
    uint16_t  RESERVED19;
    uint16_t  OR;                      //可选功能寄存器
    uint16_t  RESERVED20;
} TIM_TypeDef;
```

CMSIS 提供的主要库函数如表 7-5 所示。

表 7-5　TIM 操作库函数

函　数　名	描　　述
TIM_DeInit	将外设 TIMx 寄存器重设为默认值
TIM_TimeBaseInit	根据 TIM_TimeBaseInitStruct 中指定的参数初始化 TIMx 的时基单元
TIM_TimeBaseStructInit	根据默认值初始化 TimneBaseInitStruct 成员
TIM_OC1Init	根据 TIM_OCInitStruct 中指定的参数初始化外设 TIMx
TIM_OC2Init	根据 TIM_OCInitStruct 中指定的参数初始化外设 TIMx
TIM_OC3Init	根据 TIM_OCInitStruct 中指定的参数初始化外设 TIMx
TIM_OC4Init	根据 TIM_OCInitStruct 中指定的参数初始化外设 TIMx
TIM_OCStructInit	根据默认值初始化 TIM_OCInitStruct 成员
TIM_ICInit	根据 TIM_ICInitStruct 中指定的参数初始化外设 TIMx
TIM_PWMIConfig	根据 TIM_ICInitStruct 中的参数配置外部 PWM 测量
TIM_Cmd	使能或者失能 TIMx 外设
TIM_ITConfig	使能或者失能指定的 TIM 中断
TIM_ETRConfig	配置 TIMx 外部触发
TIM_SelectInputTrigger	选择 TIMx 输入触发源
TIM_PrescalerConfig	设置 TIMx 预分频
TIM_CounterModeConfig	设置 TIMx 计数器模式
TIM_ARRPreloadConfig	使能或失能 TIMx 在 ARR 上的预装载寄存器
TIM_OC1PreloadConfig	使能或失能 TIMx 在 CCR1 上的预装载寄存器
TIM_OC2PreloadConfig	使能或失能 TIMx 在 CCR2 上的预装载寄存器
TIM_OC3PreloadConfig	使能或失能 TIMx 在 CCR3 上的预装载寄存器

续表

函　数　名	描　　述
TIM_OC4PreloadConfig	使能或失能 TIMx 在 CCR4 上的预装载寄存器
TIM_GenerateEvent	设置 TIMx 事件由软件产生
TIM_OC1PolarityConfig	设置 TIMx 通道 1 极性
TIM_OC2PolarityConfig	设置 TIMx 通道 2 极性
TIM_OC3PolarityConfig	设置 TIMx 通道 3 极性
TIM_OC4PolarityConfig	设置 TIMx 通道 4 极性
TIM_SelectOnePulseMode	设置 TIMx 单脉冲模式
TIM_SelectOutputTrigger	选择 TIMx 触发输出模式
TIM_SelectSlaveMode	选择 TIMx 从模式
TIM_SetCounter	设置 TIMx 计数器寄存器值
TIM_SetAutoreload	设置 TIMx 自动重装载寄存器值
TIM_SetCompare1	设置 TIMx 捕获/比较 1 寄存器值
TIM_SetCompare2	设置 TIMx 捕获/比较 2 寄存器值
TIM_SetCompare3	设置 TIMx 捕获/比较 3 寄存器值
TIM_SetCompare4	设置 TIMx 捕获/比较 4 寄存器值
TIM_SetIC1Prescaler	设置 TIMx 输入/捕获 1 预分频
TIM_SetIC2Prescaler	设置 TIMx 输入/捕获 2 预分频
TIM_SetIC3Prescaler	设置 TIMx 输入/捕获 3 预分频
TIM_SetIC4Prescaler	设置 TIMx 输入/捕获 4 预分频
TIM_SetClockDivision	设置 TIMx 的时钟分割值
TIM_GetCapture1	获得 TIMx 输入/捕获 1 的值
TIM_GetCapture2	获得 TIMx 输入/捕获 2 的值
TIM_GetCapture3	获得 TIMx 输入/捕获 3 的值
TIM_GetCapture4	获得 TIMx 输入/捕获 4 的值
TIM_GetCounter	获得 TIMx 计数器的值
TIM_GetPrescaler	获得 TIMx 预分频值
TIM_GetFlagStatus	检查指定的 TIM 标志位设置与否
TIM_ClearFlag	清除 TIMx 的待处理标志位
TIM_GetITStatus	检查指定的 TIM 中断发生与否
TIM_ClearITPendingBit	清除 TIMx 的中断待处理位

1）TIM_DeInit 函数

TIM_DeInit 函数将重新启动 TIMx 控制器时钟，输入参数为 TIMx

函数原型：void TIM_DeInit(TIM_TypeDef * TIMx)

2）TIM_TimeBaseInit 函数

TIM_TimeBaseInit 函数根据 TIM_TimeBaseInitStruct 中指定的参数初始化 TIMx 的时基单元，输入参数为 TIMx，即所要初始化的定时器，函数原型如下：

```
void TIM_TimeBaseInit (TIM_TypeDef * TIMx, TIM_TimeBaseInitTypeDef *
                  TIM_TimeBaseInitStruct)
```

其中 TIM_TimeBaseInitStruct 结构体包含了 TIMx 时基单元配置参数，其定义如下

```
typedef struct
{
    uint16_t TIM_Period;
    uint16_t TIM_Prescaler;
    uint8_t TIM_ClockDivision;
    uint16_t TIM_CounterMode;
} TIM_TimeBaseInitTypeDef;
```

① TIM_Period 为自动重载寄存器值，取值范围为 0x0000～0xFFFF。

② TIM_Prescaler 为时钟的预分频值，取值范围为 0x0000～0xFFFF。

③ TIM_ClockDivision 为数字滤波时钟分频参数，即寄存器 TIMx_CR1 的 CKD 值，其取值为 TIM_CKD_DIV1、TIM_CKD_DIV2 和 TIM_CKD_DIV4，分别表示不分频、2 分频和 4 分频。

④ TIM_CounterMode 设置计数模式，取值为：

- TIM_CounterMode_Up：TIM 向上计数模式；
- TIM_CounterMode_Down：TIM 向下计数模式；
- TIM_CounterMode_CenterAligned1：TIM 中央对齐模式 1 计数模式；
- TIM_CounterMode_CenterAligned2：TIM 中央对齐模式 2 计数模式；
- TIM_CounterMode_CenterAligned3：TIM 中央对齐模式 3 计数模式。

例如，定时器 ARR 为 65535，16 分频，采样时钟不分频，向上计数的配置代码如下：

```
TIM_TimeBaseInitTypeDef   TIM_TimeBaseStructure;
TIM_TimeBaseStructure.TIM_Period = 0xFFFF;
TIM_TimeBaseStructure.TIM_Prescaler = 0xF;
TIM_TimeBaseStructure.TIM_ClockDivision = 0x0;
TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
TIM_TimeBaseInit(TIM2, & TIM_TimeBaseStructure);
```

3）TIM_OC1Init 函数

TIM_OC1Init 根据 TIM_OCInitStruct 中指定的参数初始化外设 TIMx 的输出比较通道 1，其函数原型为：

```
void TIM_OC1Init(TIM_TypeDef* TIMx, TIM_OCInitTypeDef* TIM_OCInitStruct)
```

其中，TIM_OCInitStruct 包含了 TIMx 的比较输出配置信息，其结构体定义如下：

```
typedef struct
{
  uint16_t TIM_OCMode;
  uint16_t TIM_Pulse;
  uint16_t TIM_OCPolarity;
} TIM_OCInitTypeDef;
```

① TIM_OCMode 选择定时器模式，对应于 TIMx_CCMRx 的 OCxM，取值为：

- TIM_OCMode_Timing：冻结模式，输出不起作用；
- TIM_OCMode_Active：匹配输出高电平；
- TIM_OCMode_Inactive：匹配输出低电平；
- TIM_OCMode_Toggle：匹配翻转输出；
- TIM_OCMode_PWM1：脉冲宽度调制模式 1；
- TIM_OCMode_PWM2：脉冲宽度调制模式 2。

② TIM_Pulse 设置捕获/比较寄存器的脉冲数，取值范围为 0x0000～ 0xFFFF。

③ TIM_OutputState 设置是否启用输出比较模式，TIM_OutputState_Enable 表示启用，TIM_OutputState_Disable 表示禁用。

④ TIM_OCPolarity 输出极性，取值 TIM_OCPolarity_High 表示输出信号翻转，TIM_OCPolarity_Low 表示输出信号不翻转。

示例：配置 TIM2 的输出通道 1 位 PWM 模式 1。

```
TIM_OCInitTypeDef TIM_OCInitStructure;
TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;
TIM_OCInitStructure.TIM_Pulse = 0x3FFF;
TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High;
TIM_OC1Init(TIM2, & TIM_OCInitStructure);
TIM_OC2Init、TIMOC3Init 和 TIM_OC4Init 函数与 TIM_OC1Init 函数类似。
```

4) TIM_ICInit 函数

TIM_ICInit 根据 TIM_ICInitStruct 中指定的参数初始化外设 TIMx 的输入捕获通道，其函数原型为：

```
void TIM_ICInit(TIM_TypeDef* TIMx, TIM_ICInitTypeDef* TIM_ICInitStruct)
```

其中，TIM_ICInitStruct 包含了 TIMx 的输入配置信息，其结构体的定义如下：

```
typedef struct
{
  uint16_t TIM_Channel;
  uint16_t TIM_ICPolarity;
```

```
    uint16_t TIM_ICSelection;
    uint16_t TIM_ICPrescaler;
    uint16_t TIM_ICFilter;
} TIM_ICInitTypeDef;
```

① TIM_Channel 选择通道，TIM_Channel_x 表示使用 TIM 通道 x。

② TIM_ICPolarity 输入的捕获信号沿，TIM_ICPolarity_Rising 为上升沿，TIM_ICPolarity_Falling 为下降沿。

③ TIM_ICSelection 选择输入，其取值如下：

- TIM_ICSelection_DirectTI：TIM 输入 x 与 ICx 对应相连；
- TIM_ICSelection_IndirectTI：TIM 输入 x 与 ICx 不对应，交叉相连。

④ TIM_ICPrescaler 设置输入捕获预分频器，其取值 TIM_ICPSC_DIV1、TIM_ICPSC_DIV2、TIM_ICPSC_DIV3、TIM_ICPSC_DIV4 分别表示 TIM 捕获每 1，2，3，4 个事件执行一次。

⑤ TIM_ICFilter 选择输入比较滤波器，取值范围 0x0～0xF。

示例：

```
TIM_ICInitStructure.TIM_Channel      = TIM_Channel_2;
TIM_ICInitStructure.TIM_ICPolarity  = TIM_ICPolarity_Rising;
TIM_ICInitStructure.TIM_ICSelection = TIM_ICSelection_DirectTI;
TIM_ICInitStructure.TIM_ICPrescaler = TIM_ICPSC_DIV1;
TIM_ICInitStructure.TIM_ICFilter = 0x0;
TIM_ICInit(TIM4, &TIM_ICInitStructure);
```

TIM_ICInit 的实现调用 TIx_Config 函数配置 CCER 和 CCMR1 寄存器的极性、通道和滤波，调用 TIM_SetIC1Prescaler 函数配置 CCMR1 寄存器的捕获事件分频。

5) TIM_Cmd 函数

TIM_Cmd 用来使能或停止定时器 TIMx，其函数原型为：

```
void TIM_Cmd(TIM_TypeDef * TIMx, FunctionalState NewState)
```

参数 NewState 的取值为 ENABLE 或者 DISABLE。

6) TIM _ITConfig 函数

TIM_ITConfig 用于配置 TIMx 的中断，其函数原型为：

```
void TIM_ITConfig(TIM_TypeDef * TIMx, uint16_t TIM_IT, FunctionalState NewState)
```

其中 TIM_IT 为 TIM 中断源，包括更新中断 TIM_IT_Update，捕获比较中断 TIM_IT_CC1～TIM_IT_CC4 和触发中断 TIM_IT_Trigger。

例如，启用 TIM2 的捕获比较通道 1 中断 TIM_ITConfig(TIM2, TIM_IT_CC1, ENABLE)。

7) TIM_ETRConfig 函数

TIM_ETRConfig 用于配置外部触发模式 2 下时钟的极性、预分频和滤波参数，其原

型为：

```
void TIM_ETRConfig(TIM_TypeDef * TIMx, uint16_t TIM_ExtTRGPrescaler, uint16_t TIM_ExtTRGPolarity, uint8_t ExtTRGFilter)
```

8）TIM_PrescalerConfig 函数

TIM_PrescalerConfig 函数用于配置分频系数和是否启用预装载，其函数原型为：

```
void TIM_PrescalerConfig(TIM_TypeDef * TIMx, uint16_t Prescaler,uint16_t TIM_PSCReloadMode)
```

参数 TIM_PSCReloadMode 的取值为：TIM_PSCReloadMode_Update 和 TIM_PSCReloadMode_Immediate，选用后者时，TIM 预分频值即时装入。

9）TIM_ARRPreloadConfig 函数

TIM_ARRPreloadConfig 用于配置是否启用 ARR 寄存器预装载功能，对应于 TIMx_CR1 的 ARPE，其函数原型为：

```
void TIM_ARRPreloadConfig(TIM_TypeDef * TIMx, FunctionalState Newstate)
```

参数 NewState 可以取 ENABLE 或 DISABLE。

10）TIM_OC1PreloadConfig 函数

TIM_OC1PreloadConfig 函数用于配置是否启用 OC1 寄存器的预装载功能，其函数原型为：

```
void TIM_OC1PreloadConfig(TIM_TypeDef * TIMx, uint16_t TIM_OCPreload)
```

TIM_OCPreload 参数的取值为预装载使能 TIM_OCPreload_Enable 和预装载禁用 TIM_OCPreload_Disable。

函数 TIM_OC2PreloadConfig、TIM_OC3PreloadConfig、TIM_OC4PreloadConfig 分别用于配置捕获通道 2～4。

11）TIM_SelectSlaveMode 函数

TIM_SelectSlaveMode 用于配置从模式控制器的参数，其函数原型为：

```
void TIM_SelectSlaveMode(TIM_TypeDef * TIMx, uint16_t TIM_SlaveMode)
```

TIM_SlaveMode 参数的取值包括：

- TIM_SlaveMode_Reset：触发信号（TRGI）的上升沿复位计数器并触发更新；
- TIM_SlaveMode_Gated：当触发信号（TRGI）为高电平计数器时钟使能；
- TIM_SlaveMode_Trigger：计数器在触发（TRGI）的上升沿开始计数；
- TIM_SlaveMode_External1 选中触发（TRGI）的上升沿作为计数器时钟。

12）TIM_SetCounter 函数

TIM_SetCounter 用于设置 TIMx 的计数器寄存器值，原型为：

```
void TIM_SetCounter(TIM_TypeDef * TIMx, uint16_t Counter)
```

13）TIM_ SetAutoreload 函数

TIM_ SetAutoreload 用于设置 TIMx 自动重装载寄存器值，其函数原型为：

```
Void TIM_SetAutoreload(TIM_TypeDef *  TIMx, uint16_t TIMAutoreload)
```

14）TIM_SetCompare1 函数

TIM_SetCompare1 函数用于设置 TIMx 捕获比较寄存器 1 的值，其函数原型为：

```
void TIM_SetCompare1(TIM_TypeDef *  TIMx, uint16_t Compare1)
```

TIM_SetCompare2、TIM_SetCompare3、TIM_SetCompare4 分别用于设置捕获比较寄存器 2～4。

15）TIM_SetIC1Prescaler 函数

TIM_SetIC1Prescaler 用于设置 TIMx 输入捕获 1 的预分频，其函数原型为：

```
void TIM_SetIC1Prescaler(TIM_TypeDef *  TIMx, uint16_t TIM_IC1Prescaler)
```

TIM_IC1Prescaler 的取值为 TIM_ICPSC_DIV1、TIM_ICPSC_DIV2、TIM_ICPSC_DIV4、TIM_ICPSC_DIV8，分别表示 0、2、4、8 个事件触发一次捕获。

函数 TIM_SetIC2Prescaler、TIM_SetIC3Prescaler 和 TIM_SetIC4Prescaler 分别用于输入捕获通道 2～4 的设置。

16）函数 TIM_SetClockDivision

TIM_SetClockDivision 用于设置 TIMx 的采样时钟 fdts 分割值，函数原型为：

```
void TIM_SetClockDivision(TIM_TypeDef *  TIMx, uint16_t TIM_CKD)
```

TIM_CKD 的时钟分割值取值为 TIM_CKD_DIV1、TIM_CKD_DIV2、TIM_CKD_DIV4，分别表示 CK_CNT 的 1～4 分频。

17）TIM_GetCapture1 函数

TIM_GetCapture1 用于获取捕获寄存器 1 的值，函数原型为：

```
uint16_t TIM_GetCapture1(TIM_TypeDef *  TIMx)
```

函数 TIM_GetCapture2、TIM_GetCapture3 、TIM_GetCapture4 分别表示捕获获取捕获寄存器 2～4 的值。

18）TIM_GetCounter 函数

TIM_GetCounter 用于获取定时器的计数器值，函数原型为：

```
uint16_t TIMCounter = TIM_GetCounter(TIM2);
```

19）TIM_GetPrescale 函数

TIM_GetPrescaler 用于获取定时器的时钟预分频值，函数原型为：

```
uint16_t TIM_GetPrescaler (TIM_TypeDef *  TIMx)
```

20）TIM_ GetFlagStatus 函数

TIM_ GetFlagStatus 用于获取 TIMx_SR 寄存器中指定的 TIM 标志位的值，输出结果为 SET(1)或 RESET(0)，其函数原型为：

```
FlagStatus TIM_GetFlagStatus(TIM_TypeDef * TIMx, uint16_t TIM_FLAG)
```

可获取的 TIM_FLAG 包括：

- TIM_FLAG_Update：TIM 更新标志位；
- TIM_FLAG_CCx：TIM 捕获/比较通道 x 标志位；
- TIM_FLAG_Trigger：TIM 触发标志位；
- TIM_FLAG_CCxOF：TIM 捕获/比较通道 x 溢出标志位。

21）TIM_ ClearFlag 函数

TIM_ ClearFlag 用于清除 TIMx_SR 寄存器的标志位，其函数原型为：

```
void TIM_ClearFlag(TIM_TypeDef * TIMx, uint32_t TIM_FLAG)
```

TIM_FLAG 参数的取值与 TIM_ GetFlagStatus 函数相同。

22）TIM_ GetITStatus 函数

TIM_ GetITStatus 用于检查 TIMx 的中断发生与否，其函数原型为：

```
ITStatus TIM_GetITStatus(TIM_TypeDef * TIMx, uint16_t TIM_IT)
```

TIM_IT 是待检查的 TIM 中断源，包括：TIM_IT_Update、TIM_IT_CCx 和 TIM_IT_Trigger。

23）TIM_ ClearITPendingBit 函数

TIM_ ClearITPendingBit 用于清除 TIMx 的中断待处理位，其函数原型为：

```
void TIM_ClearITPendingBit(TIM_TypeDef * TIMx, uint16_t TIM_IT)
```

TIM_IT 的取值与 TIM_GetITStatus 函数相同。

24）TIM_PWMIConfig 函数

TIM_PWMIConfig 用于根据 TIM_ICInitStruct 中的参数配置外部 PWM 输入测量，其函数原型为：

```
void TIM_PWMIConfig(TIM_TypeDef * TIMx, TIM_ICInitTypeDef * TIM_ICInitStruct)
```

7.8　定时器应用例程

7.8.1　定时器寄存器操作案例

【例 7-3】　TIM_TimeBaseInit 函数的寄存机操作实现。

```
void TIM_TimeBaseInit(TIM_TypeDef * TIMx, TIM_TimeBaseInitTypeDef * TIM_TimeBaseInitStruct)
```

```
{
    uint16_t tmpcr1 = 0;
    //获取控制寄存器 CR1 的值
    tmpcr1 = TIMx->CR1;
    //如果是通用定时器,需要配置计数模式 TIM_CR1_DIR 和 TIM_CR1_CMS
    tmpcr1 |= (uint32_t)TIM_TimeBaseInitStruct->TIM_CounterMode;
    //如果是基本定时器,只需配置 TIM_CR1_CKD
    tmpcr1 |= (uint32_t)TIM_TimeBaseInitStruct->TIM_ClockDivision;
    TIMx->CR1 = tmpcr1;                                   //将配置写入 CR1 寄存器
    TIMx->ARR = TIM_TimeBaseInitStruct->TIM_Period ;      //写入预装载值
    //写入时钟分频因子
    TIMx->PSC = TIM_TimeBaseInitStruct->TIM_Prescaler;
    //软件产生一个更新事件,将 ARR 和 PSC 的值立即装入影子寄存器
    TIMx->EGR = TIM_PSCReloadMode_Immediate;
}
```

【例 7-4】 TIM_OC1Init 的寄存器操作。

```
void TIM_OC1Init(TIM_TypeDef* TIMx, TIM_OCInitTypeDef* TIM_OCInitStruct)
{
    uint16_t tmpccmrx = 0, tmpccer = 0;
    //配置前首先禁用 CC1
    TIMx->CCER &= (uint16_t)(~(uint16_t)TIM_CCER_CC1E);
    //保存 CCER 和 CCMR1 的原始值
    tmpccer = TIMx->CCER;
    tmpccmrx = TIMx->CCMR1;
    // 配置输出比较模式
    tmpccmrx |= TIM_OCInitStruct->TIM_OCMode;
    //选择输出极性
    tmpccer |= TIM_OCInitStruct->TIM_OCPolarity;
    //设置比较输出使能
    tmpccer |= TIM_OCInitStruct->TIM_OutputState;
    //设置比较寄存器值
    TIMx->CCR1 = TIM_OCInitStruct->TIM_Pulse;
    //将配置数据写入 CCMR1 寄存器和 CCER 寄存器
    TIMx->CCMR1 = tmpccmrx;
    TIMx->CCER = tmpccer;
}
```

7.8.2 基本计时中断示例

配置一个定时器的基本流程是：

(1) 配置时钟(一般在 systeminit 中已经进行了配置);

（2）配置中断向量 NVIC_Init()；

（3）配置定时器参数 TIM_TimeBaseInit()，如果不配置，则按照默认参数 ARR＝65535，不分频，向上计数配置；

（4）开启定时器 TIM_Cmd(TIMx，ENABLE)；

（5）中断处理函数 TIMx_IRQHandler()。

【例 7-5】 配置一个通用定时器，使用溢出中断控制 LED 灯，使得 LED 灯以 500ms 的周期翻转。

分析：采用 TIM2 进行配置，TIM2 初始化时主频设定为 32MHz，因此分频设为 3200，这样一个计数为 10kHz，自动重装载值设为 5000，即可得到 500ms 定时长度。在中断产生后，通过状态寄存器的值来判断此次产生的中断属于什么类型。然后执行相关的操作，我们这里使用的是更新中断，在处理完中断之后应该向 TIM2_SR 的最低位写 0，来清除该中断标志。

在固件库函数里面，用来读取中断状态寄存器的值判断中断类型的函数是：ITStatus TIM_GetITStatus，用来判断定时器 TIMx 的中断类型 TIM_IT 是否发生中断。判断定时器 2 是否发生更新中断，方法为：

```
if(TIM_GetITStatus(TIM2,TIM_IT_Update)!=RESET){}
```

清除定时器 TIMx 的中断 TIM_IT 标志位，方法为：

```
TIM_ClearITPendingBit(TIM2,TIM_IT_Update)
```

程序代码如下：

```
void main()
{
    //开启时钟
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2,ENABLE);
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOB,ENABLE);
    //配置中断向量
    NVIC_InitTypeDef NVIC_InitStructure;
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
    NVIC_InitStructure.NVIC_IRQChannel=TIM2_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority=0;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority=0;
    NVIC_InitStructure.NVIC_IRQChannelCmd=ENABLE;
    NVIC_Init(&NVIC_InitStructure);
    //I/O初始化
    GPIO_InitTypeDef GPIO_InitStructure;
    GPIO_InitStructure.GPIO_Pin =GPIO_Pin_7;
    GPIO_InitStructure.GPIO_Speed =GPIO_Speed_40MHz;
    GPIO_InitStructure.GPIO_Mode =GPIO_Mode_Out_PP;
    GPIO_Init(GPIOB,&GPIO_InitStructure);
```

```
    //通用定时器 TIM 设置
    TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
    TIM_DeInit(TIM2);                                          //缺省 TIMER 配置
    //配置定时器参数
    TIM_InternalClockConfig(TIM2)                              //设置 TIM 时钟源为内部
    时钟
    TIM_TimeBaseStructure.TIM_Prescaler=3200-1;                //设置预分频系数
    TIM_TimeBaseStructure.TIM_ClockDivision=TIM_CKD_DIV1;//设置时钟分频
    //设置计数器计数模式为向上计数
    TIM_TimeBaseStructure.TIM_CounterMode=TIM_CounterMode_Up;
    TIM_TimeBaseStructure.TIM_Period=5000-1;                   //设置定时周期 5000
    TIM_TimeBaseInit(TIM2,&TIM_TimeBaseStructure);             //初始化设置

    TIM_ClearFlag(TIM2,TIM_FLAG_Update);                       //清除溢出中断标志
    TIM_ARRPreloadConfig(TIM2, DISABLE)                        //禁止 ARR 预装载缓冲器

    TIM_ITConfig(TIM2,TIM_IT_Update,ENABLE)                    //开启 TIM2 中断
    TIM_Cmd(TIM2,ENABLE);                                      //使能定时器 TIM2
    while(1);
}
```

【思考题：上述程序中 TIM_ClearFlag，TIM_ARRPreloadConfig 两行是否可以删掉？**】**

```
//TIM2 中断处理函数
void TIM2_IRQHandler(void)
{
    Uint8_t ReadValue;
    If(TIM_GetITStatus(TIM2, TIM_IT_Update) !=RESET)
    {
        TIM_ClearITPendingBit(TIM2, TIM_FLAG_Update);          //清除 TIM2 中断
        //读取 PB7 引脚输出数值
        ReadValue ReadValue =GPIO_ReadOutputDataBit(GPIOB,GPIO_Pin_7);
        If(ReadValue ==0)
            GPIO_SetBits(GPIOB,GPIO_Pin_7);
        else
            GPIO_ResetBits(GPIOB,GPIO_Pin_7);
    }
}
```

7.8.3 比较输出示例

比较输出的配置流程为：

(1) 启用 TIM 总线时钟；

(2) 配置输出通道的 GPIO 参数,GPIO_Init()、GPIO_PinAFConfig();
(3) 配置时基单元参数;
(4) 配置比较输出参数:方向、模式、预装载值、输出极性等;
(5) 初始化输出比较寄存器;
(6) 配置中断,使能输出比较中断;
(7) 启用定时器。

TIM 定时器输入捕获和输出比较的 I/O 引脚如表 7-6 所示。

表 7-6 TIM 对应的输入输出引脚

定时器输入输出通道	I/O 引脚
TIM2_CH1_ETR	PA0、PA5、PA15、PE9
TIM2_CH2	PA1、PB3、PE10
TIM2_CH3	PA2、PB10、PE11
TIM2_CH4	PA3、PB11、PE12
TIM3_ETR	PD2、PE2
TIM3_CH1	PA6、PB4、PC6、PE3
TIM3_CH2	PA7、PB5、PC7、PE4
TIM3_CH3	PB0、PC8
TIM3_CH4	PB1、PC9
TIM4_ETR	PE0
TIM4_CH1	PB6、PD12
TIM4_CH2	PB7、PD13
TIM4_CH3	PB8、PD14
TIM4_CH4	PB9、PD15
TIM9_CH1	PB13、PD0、PE5
TIM9_CH2	PB14、PD7、PE6
TIM10_CH1	PA6、PB8、PB12、PE0
TIM10_CH2	PA7
TIM11_CH1	PB9、PB15、PE1

【例 7-6】 APB1 总线主频设为 8MHz,配置定时器 TIM3,通过定时器 TIM3 的 4 个比较中断通道在 4 个 I/O 口上输出 0.5Hz、1Hz、2Hz 和 4Hz 不同频率的方波,控制 LED 灯。

分析:设定总线主频为 8MHz,此时若采用内部时钟,则驱动定时器的时钟为 16MHz,预分频设为 16000,则一个计数周期为 1ms,这样我们通过给定比较值 1000、500、250 和 125 即可产生 4 个比较中断,通过输出比较通道控制电平进行翻转。

```
#include "stm32L1xx.h"
#include "stm32L1xx_gpio.h"
#include "stm32L1xx_tim.h"
TIM_TimeBaseInitTypeDef  TIM_TimeBaseStructure;
GPIO_InitTypeDef GPIO_InitStructure;
NVIC_InitTypeDef NVIC_InitStructure;
TIM_OCInitTypeDef  TIM_OCInitStructure;
//比较寄存器值
uint16_t CCR1_Val = 1000;
uint16_t CCR2_Val = 500;
uint16_t CCR3_Val = 250;
uint16_t CCR4_Val = 125;
uint16_t PrescalerValue = 0;

int main(void)
{
    //配置总线时钟为 HCLK/4,此时定时器时钟为 HCLK/2 = 16MHz
    RCC_PCLK1Config(RCC_HCLK_Div4);
    //开启 TIM 和 GPIO 时钟
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOB, ENABLE);
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE);
    //TIM3 中断配置
    NVIC_InitStructure.NVIC_IRQChannel = TIM3_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
    //GPIO 配置 A6、A7 推挽输出模式,上拉
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6|GPIO_Pin_7 ;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_40MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
    GPIO_PinAFConfig(GPIOA, GPIO_PinSource6, GPIO_AF_TIM3);
    GPIO_PinAFConfig(GPIOA, GPIO_PinSource7, GPIO_AF_TIM3);
    //GPIO 配置 B0 和 B1 推挽输出模式
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0|GPIO_Pin_1 ;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_40MHz;
    GPIO_Init(GPIOB, &GPIO_InitStructure);
```

```
    GPIO_PinAFConfig(GPIOB, GPIO_PinSource0, GPIO_AF_TIM3);
    GPIO_PinAFConfig(GPIOB, GPIO_PinSource1, GPIO_AF_TIM3);
    //时基参数配置
    PrescalerValue = 16000 - 1;
    TIM_TimeBaseStructure.TIM_Period = 65535;
    TIM_TimeBaseStructure.TIM_Prescaler = PrescalerValue;
    TIM_TimeBaseStructure.TIM_ClockDivision = 0x0;
    TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
    TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStructure);
    //输出比较通道 1 设置
    TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
    TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_Toggle;
    TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_Low;
    TIM_OCInitStructure.TIM_Pulse = CCR1_Val;
    TIM_OC1Init(TIM3, &TIM_OCInitStructure);
    TIM_OC1PreloadConfig(TIM3, TIM_OCPreload_Disable);
    //输出比较通道 2 设置
    TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
    TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_Toggle;
    TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_Low;
    TIM_OCInitStructure.TIM_Pulse = CCR2_Val;
    TIM_OC2Init(TIM3, &TIM_OCInitStructure);
    TIM_OC2PreloadConfig(TIM3, TIM_OCPreload_Disable);
    //输出比较通道 3 设置
    TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
    TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_Toggle;
    TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_Low;
    TIM_OCInitStructure.TIM_Pulse = CCR3_Val;
    TIM_OC3Init(TIM3, &TIM_OCInitStructure);
    TIM_OC3PreloadConfig(TIM3, TIM_OCPreload_Disable);
    //输出比较通道 4 设置
    TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
    TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_Toggle;
    TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_Low;
    TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
    TIM_OCInitStructure.TIM_Pulse = CCR4_Val;
    TIM_OC4Init(TIM3, &TIM_OCInitStructure);
    TIM_OC4PreloadConfig(TIM3, TIM_OCPreload_Disable);
    //输出比较中断使能
    TIM_ITConfig(TIM3,TIM_IT_CC1|TIM_IT_CC2|TIM_IT_CC3|TIM_IT_CC4, ENABLE);
    TIM_Cmd(TIM3, ENABLE);                                    //启动定时器
    while (1);
}
```

stm32l1xx_it.c中断函数

```
uint16_t capture = 0;
extern uint16_t CCR1_Val;
extern uint16_t CCR2_Val;
extern uint16_t CCR3_Val;
extern uint16_t CCR4_Val;

void TIM3_IRQHandler(void)
{
   if (TIM_GetITStatus(TIM3, TIM_IT_CC1) != RESET)
   { //比较中断 1,翻转 PA6,重设比较值
     TIM_ClearITPendingBit(TIM3, TIM_IT_CC1);
     capture = TIM_GetCapture1(TIM3);
     TIM_SetCompare1(TIM2, capture + CCR1_Val);
   }
   else if (TIM_GetITStatus(TIM3, TIM_IT_CC2) != RESET)
   { //比较中断 2,翻转 PA7,重设比较值
     TIM_ClearITPendingBit(TIM3, TIM_IT_CC2);

     capture = TIM_GetCapture2(TIM3);
     TIM_SetCompare2(TIM3, capture + CCR2_Val);
   }
   else if (TIM_GetITStatus(TIM3, TIM_IT_CC3) != RESET)
   { //比较中断 3,翻转 PB0,重设比较值
     TIM_ClearITPendingBit(TIM3, TIM_IT_CC3);
     capture = TIM_GetCapture3(TIM2);
     TIM_SetCompare3(TIM3, capture + CCR3_Val);
   }
   else
   { //比较中断 4,翻转 PB1,重设比较值
     TIM_ClearITPendingBit(TIM3, TIM_IT_CC4);
     capture = TIM_GetCapture4(TIM3);
     TIM_SetCompare4(TIM3, capture + CCR4_Val);
   }
}
```

7.8.4 输入捕获示例

输入捕获的配置流程如下：

(1) 配置 TIM 时钟；

(2) 配置 TIM 输入端口的 GPIO 参数 GPIO_Init()、GPIO_PinAFConfig()；

（3）配置时基参数 ARR、PSR，计数模式、采样频率 TIM_TimeBaseInit()；

（4）配置输入通道参数通道号、极性、捕获模式、分频以及滤波 TIM_ICInit()；

（5）配置 NVIC 中断和 DMA ：NVIC_Init()；

（6）启动定时器 TIM_Cmd(TIMx，ENABLE)；

（7）中断处理中读取捕获值 TIM_GetCapturex()。

【例 7-7】 利用 TIM3 产生一个 1kHz 固定频率的比较输出，将 TIM3 的输出通道通过外部引脚连接到 TIM4 的输入引脚进行二次捕获，然后计算频率，频率＝计数器频率/(两次捕获之间的计数器差值)，由于计数器会溢出，因此可能存在第二次捕获的捕获寄存器值小于第一次，此时需要在结果上加上预装载寄存器 ARR 的值。TIM4 输入捕获的配置代码如下：

```
void TIM_Config(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    NVIC_InitTypeDef NVIC_InitStructure;
    TIM_ICInitTypeDef   TIM_ICInitStructure;
    //使能 TIM4 和输入通道 PB7 的 GPIO 时钟
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM4, ENABLE);
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOB, ENABLE);
    //TIM4 捕获通道 2 的 GPIO 端口配置
    GPIO_InitStructure.GPIO_Mode   = GPIO_Mode_AF;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_40MHz;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_PuPd   = GPIO_PuPd_UP;
    GPIO_InitStructure.GPIO_Pin    = GPIO_Pin_7;
    GPIO_Init(GPIOB, &GPIO_InitStructure);
    GPIO_PinAFConfig(GPIOB, GPIO_PinSource7, GPIO_AF_TIM4);
    //TIM4 TI2 输入捕获,上升沿有效,不分频不滤波
    TIM_ICInitStructure.TIM_Channel      = TIM_Channel_2;
    TIM_ICInitStructure.TIM_ICPolarity   = TIM_ICPolarity_Rising;
    TIM_ICInitStructure.TIM_ICSelection = TIM_ICSelection_DirectTI;
    TIM_ICInitStructure.TIM_ICPrescaler = TIM_ICPSC_DIV1;
    TIM_ICInitStructure.TIM_ICFilter = 0x0;
    TIM_ICInit(TIM4, &TIM_ICInitStructure);
    //配置 TIM4 中断
    NVIC_InitStructure.NVIC_IRQChannel = TIM4_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
    //开启定时器,使能通道 2 输入捕获中断,由于没有配置定时器的 ARR 等值,因此采用默认值,即
    //CK_INT 作为时钟源,向上计数,最大计数值为 65535
```

```
    TIM_Cmd(TIM4, ENABLE);
    TIM_ITConfig(TIM4, TIM_IT_CC2, ENABLE);
}

//主程序
int main(void)
{
    //配置 TIM4
    TIM_Config();
    while (1);
    return 0;
}
```

中断处理程序如下：

```
uint16_t IC4ReadValue1 = 0, IC4ReadValue2 = 0;
uint16_t CaptureNumber = 0;
uint32_t Capture = 0;
uint32_t TIM4Freq = 0;
void TIM4_IRQHandler(void)
{
    if (TIM_GetITStatus(TIM4, TIM_IT_CC2) != RESET)
    { //记录第一次捕获中断 CCR 值
      TIM_ClearITPendingBit(TIM4, TIM_IT_CC2);
      if(CaptureNumber == 0)
      {
        IC4ReadValue1 = TIM_GetCapture2(TIM4);
        CaptureNumber = 1;
      }
      else if(CaptureNumber == 1)
      { //记录第二次捕获 CCR 值
        IC4ReadValue2 = TIM_GetCapture2(TIM4);
        //计算差值,如果第二次的 CCR 值小于第一次,则结果加上 65535
        if (IC4ReadValue2 > IC4ReadValue1)
          Capture = (IC4ReadValue2 - IC4ReadValue1) - 1;
        else if (IC4ReadValue2 < IC4ReadValue1)
          Capture = ((0xFFFF - IC4ReadValue1) + IC4ReadValue2) - 1;
        else
            Capture = 0;
        //计算频率,频率=主频/捕获差值
        TIM4Freq = (uint32_t) SystemCoreClock / Capture;
        CaptureNumber = 0;
      }
    }
}
```

7.8.5　PWM 输出和输入示例

PWM 的输出配置和比较输出模式一样，区别在于输出比较模式不同，且必须开启预装载功能。

【例 7-8】 配置 TIM11 输出一个占空比为 50%，频率为 50kHz 的 PWM 波。

分析：设置 TIM11 的时钟为内部时钟 32MHz，预分频为 0，这样定时器的周期 = 32M/50k = 640，因此 ARR = 639，占空比 = （TIM11_CCR1/（TIM11_ARR + 1）），因此 CCR1 设为 320 即可产生一个 50% 占空比的方波，代码如下：

```
TIM_TimeBaseInitTypeDef   TIM_TimeBaseStructure;
GPIO_InitTypeDef GPIO_InitStructure;
TIM_OCInitTypeDef   TIM_OCInitStructure;
uint16_t CCR1Val = 320;
int main(void)
{
   //使能定时器和输出引脚 GPIO 时钟
   RCC_APB2PeriphClockCmd(RCC_APB2Periph_TIM11, ENABLE);
   RCC_AHBPeriphClockCmd( RCC_AHBPeriph_GPIOB, ENABLE);
   //配置 PWM 输出引脚的 I/O 参数,并映射到 TIM11 的输出通道 1
   GPIO_InitStructure.GPIO_Mode   = GPIO_Mode_AF;
   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_40MHz;
   GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
   GPIO_InitStructure.GPIO_PuPd   = GPIO_PuPd_UP;
   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
   GPIO_Init(GPIOB, &GPIO_InitStructure);
   GPIO_PinAFConfig(GPIOB, GPIO_PinSource9, GPIO_AF_TIM11);
   //TIM11 时基单元参数配置
   TIM_TimeBaseStructure.TIM_Period = 639;
   TIM_TimeBaseStructure.TIM_Prescaler = 0;
   TIM_TimeBaseStructure.TIM_ClockDivision = 0;
   TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
   TIM_TimeBaseInit(TIM11, &TIM_TimeBaseStructure);

   //配置输出通道 1 为 PWM1 模式
   TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;
   TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
   TIM_OCInitStructure.TIM_Pulse = CCR1Val;
   TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High;
   TIM_OC1Init(TIM11, &TIM_OCInitStructure);
   TIM_OC1PreloadConfig(TIM11, TIM_OCPreload_Enable);
   TIM_ARRPreloadConfig(TIM11, ENABLE);
```

```
    //启动定时器
    TIM_Cmd(TIM11, ENABLE);
    while (1){}
}
```

【例 7-9】 利用上例中的 TIM11 输出的 PWM,将其输出引脚接到 TIM3 的输入通道 2 上,对输入 PWM 进行频率和占空比测量。

分析:TIM3 的通道 1 和通道 2 同时对 PWM 信号进行捕获,频率=TIM3 频率/TIM3_CCR2 的值,占空比=(TIM3_CCR1 * 100)/(TIM3_CCR2)。

在 PWM 输入模式下,输入配置需要使用 TIM_PWMIConfig 函数对 TIM_ICInitStructure 进行配置。代码如下:

```
TIM_ICInitTypeDef TIM_ICInitStructure;
GPIO_InitTypeDef GPIO_InitStructure;
NVIC_InitTypeDef NVIC_InitStructure;
int main(void)
{
    //TIM3 和输入捕获引脚 GPIO 时钟开启
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE);
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA, ENABLE);
    //PA7 捕获引脚 GPIO 参数配置
    GPIO_InitStructure.GPIO_Pin =   GPIO_Pin_7;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_UP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_40MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
    GPIO_PinAFConfig(GPIOA, GPIO_PinSource7, GPIO_AF_TIM3);
    // TIM3 中断配置
    NVIC_InitStructure.NVIC_IRQChannel = TIM3_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
    //TIM3 PWM 输入模式配置,通道 2,上升沿,CCR2 计算频率,CCR1 计算占空比,
    //无分频、无滤波
    TIM_ICInitStructure.TIM_Channel = TIM_Channel_2;
    TIM_ICInitStructure.TIM_ICPolarity = TIM_ICPolarity_Rising;
    TIM_ICInitStructure.TIM_ICSelection = TIM_ICSelection_DirectTI;
    TIM_ICInitStructure.TIM_ICPrescaler = TIM_ICPSC_DIV1;
    TIM_ICInitStructure.TIM_ICFilter = 0x0;
    //使用 PWMIConfig 函数初始化
    TIM_PWMIConfig(TIM3, &TIM_ICInitStructure);
```

```
    //选择触发源为 TI2
    TIM_SelectInputTrigger(TIM3, TIM_TS_TI2FP2);
    //选择从模式控制为复位模式
    TIM_SelectSlaveMode(TIM3, TIM_SlaveMode_Reset);
    //使能主从模式
    TIM_SelectMasterSlaveMode(TIM3, TIM_MasterSlaveMode_Enable);
    TIM_Cmd(TIM3, ENABLE);                                    //启用定时器
    TIM_ITConfig(TIM3, TIM_IT_CC2, ENABLE);                   //开启 TIM3 捕获 2 中断
    while (1);
}
```

中断处理函数为：

```
void TIM3_IRQHandler(void)
{
    //清除捕获中断
    TIM_ClearITPendingBit(TIM3, TIM_IT_CC2);
    //读取通道 1 和通道 2 的捕获值
    IC2Value = TIM_GetCapture2(TIM3);
    if (IC2Value != 0)
    {
      //占空比计算
      DutyCycle = (TIM_GetCapture1(TIM3) * 100) / IC2Value;
      //频率计算
      Frequency = SystemCoreClock / IC2Value;
    }
    else
    {
      DutyCycle = 0;
      Frequency = 0;
    }
}
```

Chapter 8

第 8 章　USART 串口控制器

【导读】 USART 是嵌入式系统中串行通信的常用部件,本章首先介绍串行输入输出的基本概念,同步串行和异步串行各自的特点,然后对 STM32L152 的 USART 串行接口控制器的内部结构,寄存器以及发送和接收配置流程进行介绍,最后介绍 CMSIS 提供的典型寄存器操作库函数。针对串口数据帧的传输,本章还对 HDLC 链路协议的帧构造,Modubs 通信进行介绍,以异步串口 PC 连接通信、状态机数据发送和接收为案例介绍 USART 库函数的使用方法。

8.1　串行输入输出接口的基本概念

计算机与外部数据交互的方法有并行传输和串行传输两种,并行传输一次可以传输多个二进制位,需要多根数据线同时进行传输,串行传输一次只能传输一个二进制位,数据需要一位一位地传输。一般来讲,并行传输的吞吐量高,例如计算机系统的内存接口总线等,但并行传输对线路之间的抗干扰能力要求较高,当时钟速率越高,线之间的距离越近时,线间的串扰和线上的延迟对数据传输性能都会产生影响,占用的微处理器口线也较多。而串行传输时,线路比较简单,出错时只需要传输若干位即可,适合远距离传输,且在实现上更容易提高时钟速率,获得较好的通信带宽,因此目前计算机系统里使用的 USB、SPI、SATA、网卡等高速设备均采用串行总线进行通信。

串行数据的传输分为三种模式:

- 单工传输:单工传输下一个设备只能发送或者接收;
- 半双工传输:半双工模式下,通信双方都可以进行发送和接收,但一台设备不能同时发送和接收;
- 双工传输:双工模式下传输效率最高,可以同时进行双向通信。

数字传输用 0 和 1 表示信号,传输距离比较短,且容易受到干扰,因此在数据传输时一般将数字型号调制成模拟信号,在接收端再将模拟信号解调成数字信号,这个功能由调制解调器完成,调制方法包括调幅 ASK、调频 FSK、调相 PSK 等,通过调制,一方面可以用一个模拟符号表示多个二进制位从而提高数据传输速率,另外还可以通过纠错编码提高传输可靠性。调制完的模拟信号具有不同的电气特征。以串行接口的电气标准分,串行接口包括 RS-232-C、RS-422、RS485、USB 等。RS-232-C、RS-422 与 RS-485 标准只对接口的电气特

性做出规定，不涉及接插件、电缆或协议。USB 是近几年发展起来的新型接口标准，主要应用于高速数据传输领域。

RS-232-C：也称标准串口，是目前最常用的一种串行通信接口。它是在 1970 年由美国电子工业协会(EIA)联合调制解调器、计算机终端生产厂家共同制定的用于串行通讯的标准。它的全名是“数据终端设备(DTE)和数据通信设备(DCE)之间串行二进制数据交换接口技术标准”。传统的 RS-232-C 接口标准有 22 根线，采用标准 25 芯 D 型插头座。自 IBM PC/AT 开始使用简化了的 9 芯 D 型插座，RS-232-C 曾是 PC 的标配接口，目前已被 PC 淘汰。

RS-422：为改进 RS-232 通信距离短、速率低的缺点，RS-422 定义了一种平衡通信接口，将传输速率提高到 10Mb/s，传输距离延长到 4000 英尺(速率低于 100kb/s 时)，并允许在一条平衡总线上连接最多 10 个接收器。RS-422 是一种单机发送、多机接收的单向、平衡传输规范，被命名为 TIA/EIA-422-A 标准。

RS-485：为扩展应用范围，EIA 又于 1983 年在 RS-422 基础上制定了 RS-485 标准，增加了多点、双向通信能力，即允许多个发送器连接到同一条总线上，同时增加了发送器的驱动能力和冲突保护特性，扩展了总线共模范围，后命名为 TIA/EIA-485-A 标准。

Universal Serial Bus(通用串行总线 USB)：USB 接口是电脑主板上的一种四针接口，其中中间两个针传输数据，两边两个针给外设供电。USB 接口速度快、连接简单、不需要外接电源，传输速度 12Mb/s，USB 2.0 可达 480Mb/s；电缆最大长度 5 米，USB 电缆有 4 条线：2 条信号线，2 条电源线，可提供 5V 电源；USB 通过串联方式最多可串接 127 个设备；支持热插拔。最新的规格是 USB 3.1。因此在嵌入式系统 RS-232 串口与 PC 通信时，一般采用 USB 转串口的设备进行转接。

RJ-45 接口：是以太网最为常用的接口，RJ-45 是一个常用名称，指的是由 IEC(60)603-7 标准化，使用由国际性的接插件标准定义的 8 个位置(8 针)的模块化插孔或者插头。

计算机的串行总线控制器输入和输出是数字信号，因此在使用 RS-232、RS-485 或 RJ-45 等接口时，需要外加物理层芯片进行信号转换(如 MAX232)。例如，TTL 高电平 1，电压≥2.4V，低电平 0，电压≤0.5V(对于 5V 或 3.3V 电源电压)；RS-232 采用的是负逻辑，高电平 1，电压－15V～－3V，低电平 0，电压＋3V～＋15V；TTL 电平以电源为参考，高电平 1，电压≥0.7 * VCC，低电平 0，电压≤0.2 * VCC。

8.2　串行通信协议

串行数据传输主要采用两种传输协议：同步传输协议和异步传输协议。

8.2.1　异步串行通信协议

异步通信是我们最常采用的通信方式，异步通信采用固定的通信格式，数据以相同的帧

格式传送。如图 8-1 所示，每一帧由起始位、数据位、奇偶校验位和停止位组成。

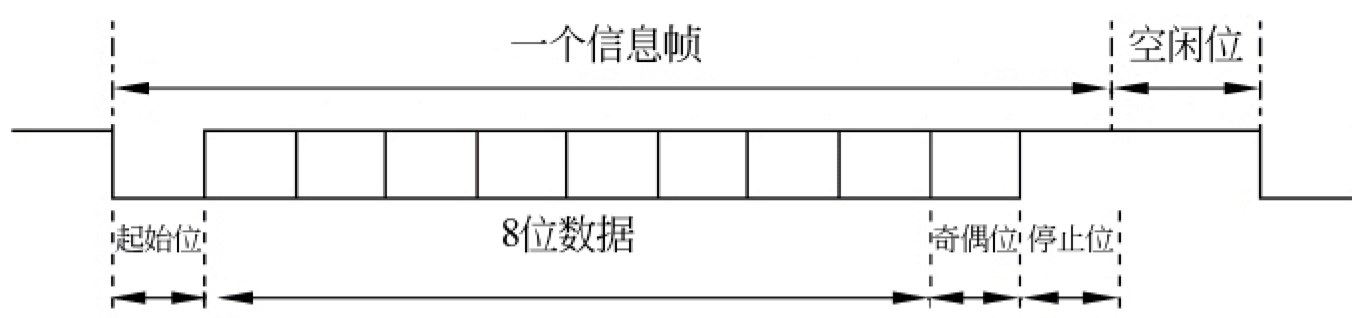


图 8-1　异步串行数据格式

在通信线上没有数据传送时处于逻辑 1 状态(高电平)。当发送设备发送一个字符数据时，首先发出一个逻辑 0 信号(低电平)，这个低电平就是起始位。起始位通过通信线传向接收设备，当接收设备检测到这个低电平后，就开始准备接收数据信号。因此，起始位所起的作用就是表示字符传送开始。起始位后面紧接着的是数据位，它可以是 5 位、6 位、7 位或 8 位。数据传送时，低位在前。奇偶校验位用于数据传送过程中的数据检错，数据通信时通信双方必须约定一致的奇偶校验方式。奇偶校验位是冗余位，可以不要校验位。在奇偶校验位或数据位后紧接的是停止位，停止位可以是 1 位、也可以是 1.5 位或 2 位。接收端收到停止位后，知道上一字符已传送完毕，同时，也为接收下一字符做好准备。若停止位后不是紧接着传送下一个字符，则让线路保持为逻辑 1。逻辑 1 表示空闲，线路处于等待状态。

8.2.2　同步串行通信协议

同步通信时，通信双方共用一个时钟，这是同步通信区分于异步通信的最显著的特点。在异步通信中，每个字符要用起始位和停止位作为字符开始和结束的标志，以致占用了部分时间。所以在数据块传送时，为提高通信速度，常去掉这些标志，而采用同步通信。同步通信中，数据开始传送前用同步字符(通常为 1～2 个特殊字符)来指示，并由时钟来实现发送端和接收端的同步，即检测到规定的同步字符后，下面就连续按顺序传送数据，直到一块数据传送完毕。同步传送时，字符之间没有间隙，也不要起始位和停止位，仅在数据开始时用同步字符 SYNC 来指示，其数据格式如图 8-2 所示。



图 8-2　同步串行数据格式

同步通信和异步通信相比，以同步字符作为传送的开始，从而使收发双方取得同步，每位占用的时间相等，字符数据之间不允许有空位，当线路空闲或没字符可发时，发送同步字符。在同步传送时，要求用时钟来实现发送端和接收端之间的同步。为了保证接收正确无误，发送方除了传送数据外，还要传送同步时钟。同步串行通信虽然可以提高传送速度，但实现起来较为复杂。

8.2.3　串行通信基本概念

1. 波特率

波特率(Baud Rate)是指数据传送时,每秒传送数据二进制代码的位数,它的单位是位/秒(b/s)。1 波特就是一位每秒。假设数据传送速率是每秒 120 个字符,而每个字符格式包括 10 个二进制位(1 个起始位、一个终止位、8 个数据位),这时传送的波特率为:10×120=1200b/s。位传送时间宽度 Td=波特率的倒数,则上式中的 Td=1/1200s=0.883ms。

在异步串行通信中,接收设备和发送设备无需传输时钟,但双方必须用各自的时钟保持相同的传送波特率,并以每个字符数据的起始位与发送设备保持同步。起始位、数据位、奇偶位和停止位的约定,在同一次传送过程中必须保持一致,这样才能成功的传送数据。

2. 接收/发送时钟

二进制数据系列在串行传送过程中以数字信号波形的形式出现。不论接收还是发送,都必须有时钟信号对传送的数据进行定位。接收/发送时钟就是用来控制通信设备接收/发送字符数据速度的,该时钟信号通常由外部时钟电路产生。

在发送数据时,发送器在发送时钟的下降沿将移位寄存器的数据串行移位输出;在接收数据时,接收器在接收时钟的上升沿对接收数据采样,进行数据位检测,接收/发送时钟频率与波特率有如下关系:收/发时钟频率=n×收/发波特率,n=1,8,16,64,时钟周期 Tc=Td/n。

在同步传送方式,必须取 n=1,即接收/发送时钟的频率等于收/发波特率。在异步传送方式,n 可配置为 8、16 或 64,即可以选择接收/发送时钟频率是波特率的 1、16、64 倍。因此可由要求的传送波特率及所选择的倍数 n 来确定接收/发送时钟的频率,如图 8-3 所示。

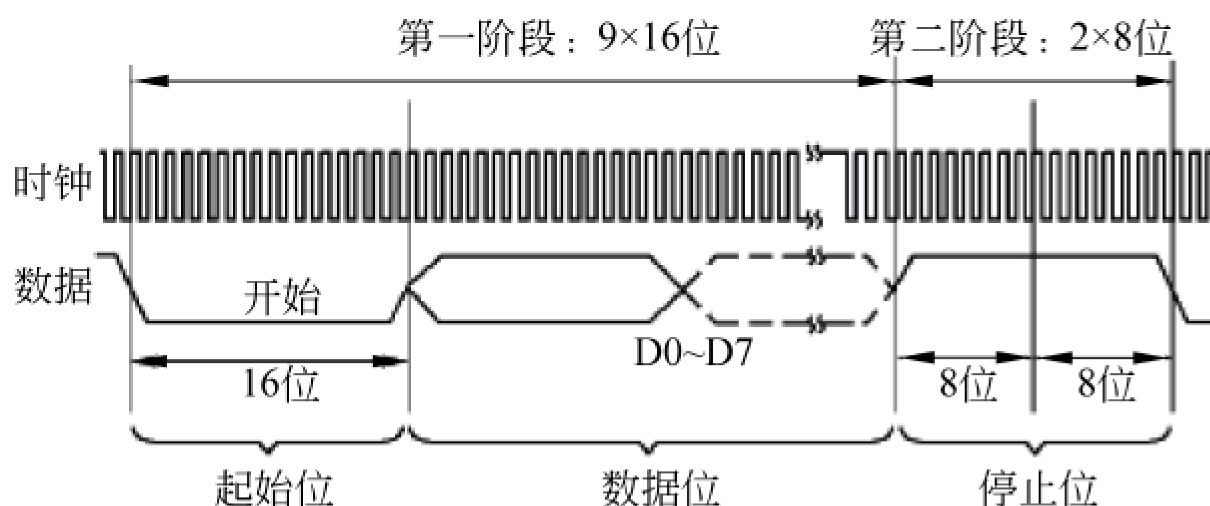


图 8-3　发送和接收时钟及波特率的关系

若取 n=16,那么异步传送接收数据实现同步的过程如下:接收器在每一个接收时钟的上升沿采样接收数据线,当发现接收数据线出现低电平时就认为是起始位的开始,以后若在连续测 8 个时钟周期(因 n=16,故 Td=16 * Tc)内检测到接收数据线仍保持低电平,则确定它为起始位(不是干扰信号)。通过这种方法,不仅能够排除接收线上的噪声干扰,识别假起始位,而且能够相当精确的确定起始位的中间点,从而提供一个正确的时间基准。从这个基准算起,每隔 16 * Tc 采样一次数据线,作为输入数据。一般来说,从接收数据线检测到

一个下降沿开始，若其低电平能保持 $n*Tc/2$（半位时间），则确定为起始位，其后每隔 $n*Tc$ 时间（一个数据时间）在每个数据位的中间点采样，时序如图 8-4 所示。

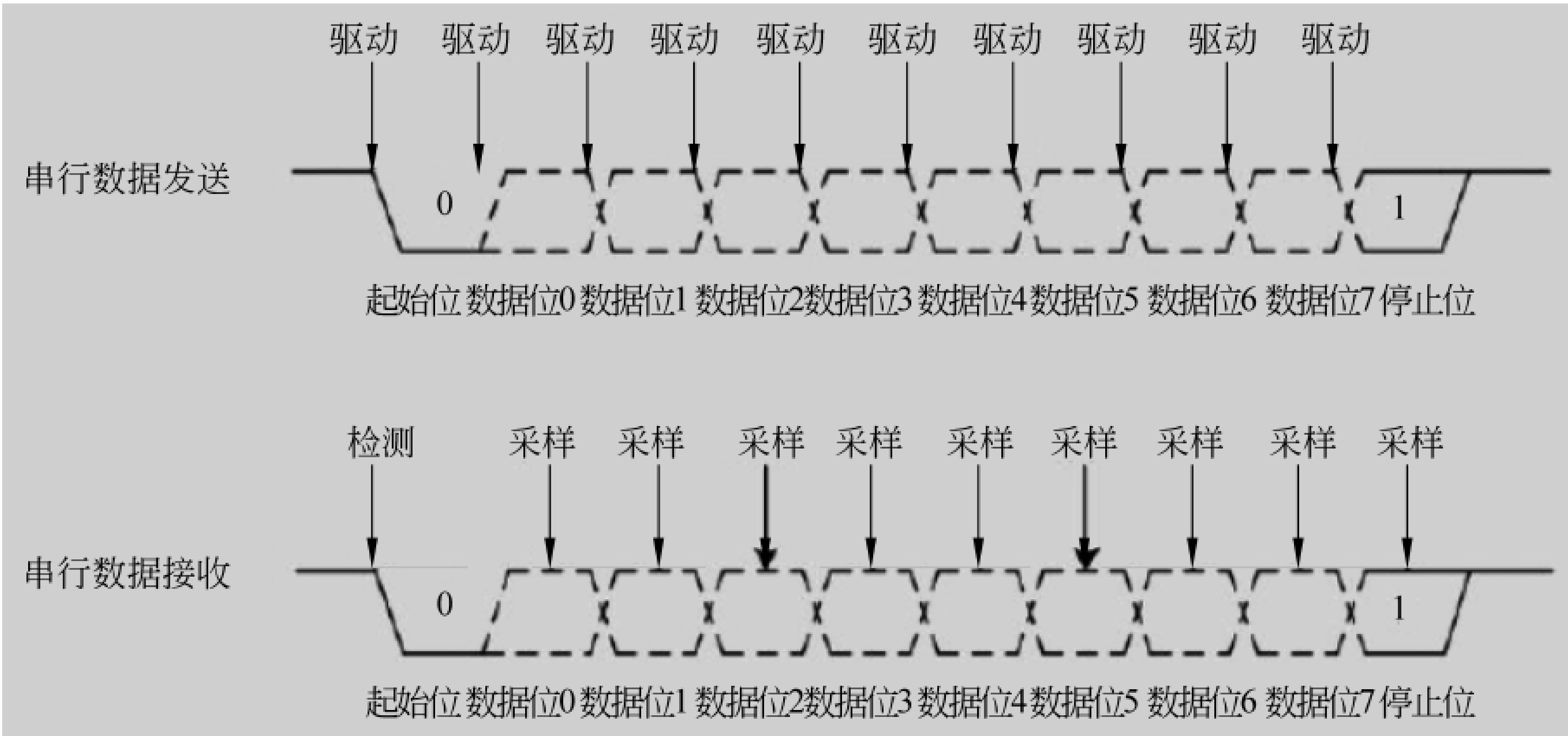


图 8-4　数据发送和采样时钟的驱动时机

3. 数据收发过程

发送数据过程：

- 空闲状态，线路处于高电位；
- 当收到发送数据指令后，拉低线路一个数据位的时间 T；
- 数据按低位到高位依次发送。

数据发送完毕后，接着发送奇偶校验位和停止位（停止位为高电位）。

接收数据过程：

- 空闲状态，线路处于高电位；当检测到线路的下降沿时说明线路有数据传输；
- 按照约定的波特率从低位到高位接收数据；
- 数据接收完毕后，接着接收并比较奇偶校验位是否正确；
- 如果正确则通知后续设备准备接收数据或存入缓存。

4. 空闲帧和断开帧

空闲帧和断开帧如图 8-5 所示。

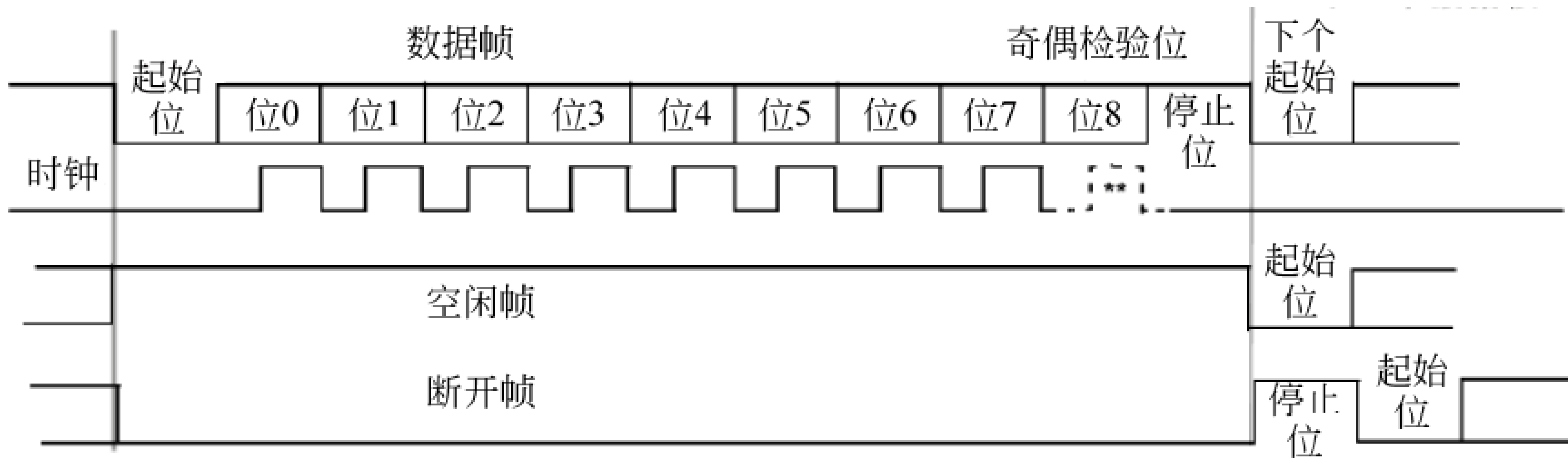


图 8-5　空闲帧和断开帧

- 空闲帧是完全由 1 组成的一个完整的数据帧，后面跟着包含了数据的下一帧的开始位（其中 1 的位数包括了停止位的位数）。
- 断开帧是在一个帧周期内全部收到 0（包括停止位期间也是 0）。在断开帧结束时，发送器可以再插入 1 或 2 个停止位来发起或接收下一个起始位。

5. 硬件流控制

数据在两个串口之间传输时，常常会出现丢失数据的现象，或者两台计算机的处理速度不同，如台式机与单片机之间的通信，接收端数据缓冲区已满，则此时继续发送来的数据就会丢失。流控制能解决这个问题，当接收端数据处理不过来时，就发出“不再接收”的信号，发送端就停止发送，直到收到“可以继续发送”的信号再发送数据。因此流控制可以控制数据传输的进程，防止数据的丢失。两种常用的流控制是硬件流控制和软件流控制。如果 UART 只有 RX、TX 两个信号，要流控的话只能是软件流控，我们在普通的控制通信中一般不用硬件流控制，而用软件流控制。一般通过 xon/xoff 两个特殊字符来实现软件流控制。常用方法是：当接收端的输入缓冲区内数据量超过设定的高位时，就向数据发送端发出特殊字符 xoff（ASCII 码 19，表示 control＋s，也可自定义），发送端收到 xoff 字符后就立即停止发送数据；当接收端的输入缓冲区内数据量低于设定的低位时，就向数据发送端发出特殊字符 xon（ASCII 码 17 或 control＋q，也可自定义），发送端收到 xon 字符后就立即开始发送数据。但是在二进制数据传输中，标志字符也有可能在数据流中出现从而引起误操作，因此可以采用硬件流控制解决。硬件流控制常用的有 RTS/CTS（请求发送/清除发送）流控制和 DTR/DSR（数据终端就绪/数据设置就绪）流控制，常用的方式是 RTS/CTS 硬件流控制。

【**思考题**：如何解决软件流控制中传输的数据内部出现 xon 或 xoff 字符的问题？】

RTS（Require To Send，请求发送）为输出信号，用于指示本设备准备好可接收数据，低电平有效，低电平说明本设备可以接收数据，由接收模块向外发出。

CTS（Clear To Send，发送允许）为输入信号，用于判断是否可以向对方发送数据，低电平有效，低电平说明本设备可以向对方发送数据，若是高电平，在当前数据传输结束时阻断下一次的数据发送，由发送模块接收此信号。

两个 USART 设备进行连接时，CTS 和 RTS 进行交叉连接，如图 8-6 所示。如果不使用 USART 的内部硬件流控制模块进行 CTS 和 RTS 控制，我们也可以通过 GPIO 模拟 RTS 和 CTS。

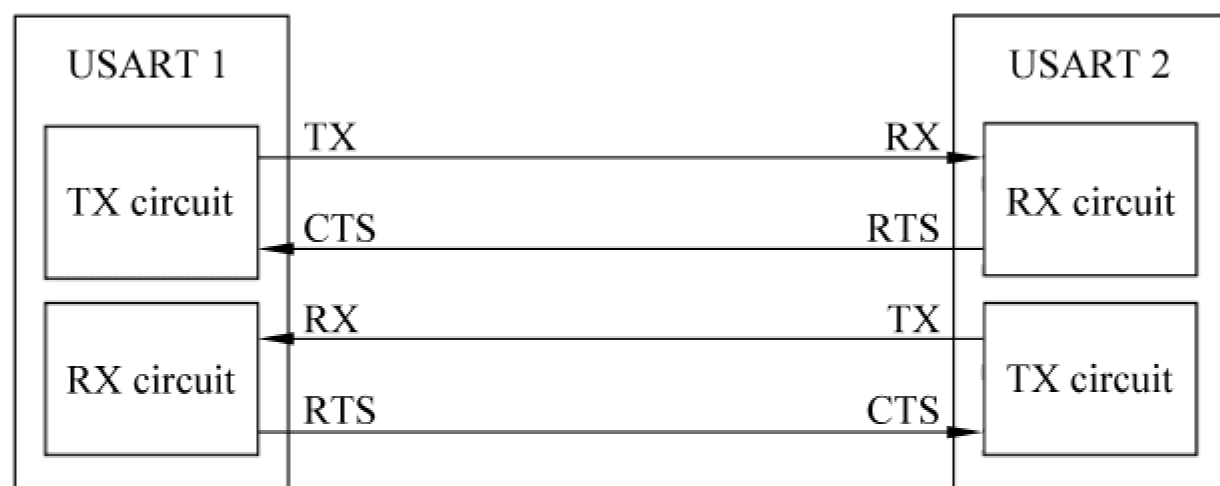


图 8-6　RTS/CTS 硬件流控制连接方式

6. 异步串行通信控制器 UART 的基本结构

如图 8-7 所示，一个异步串行总线控制器由波特率发生器、发送和接收数据控制单元以及串行并行转换单元、中断控制器等几个基本组件组成。波特率发生器用于产生发送和采样时钟，配置数据传输速率，TX 发送单元用于从总线写数据到 UART 进行发送，RX 接收单元用于将接收到的数据传输到总线。串并行转换是两个移位寄存器，用于将一个字符数据逐位发送到 TX 引脚或从 RX 引脚将每个位移位构成一个字符数据。中断控制用于将 UART 传输控制过程中的事件或错误引起的中断发送到 NVIC 控制器。

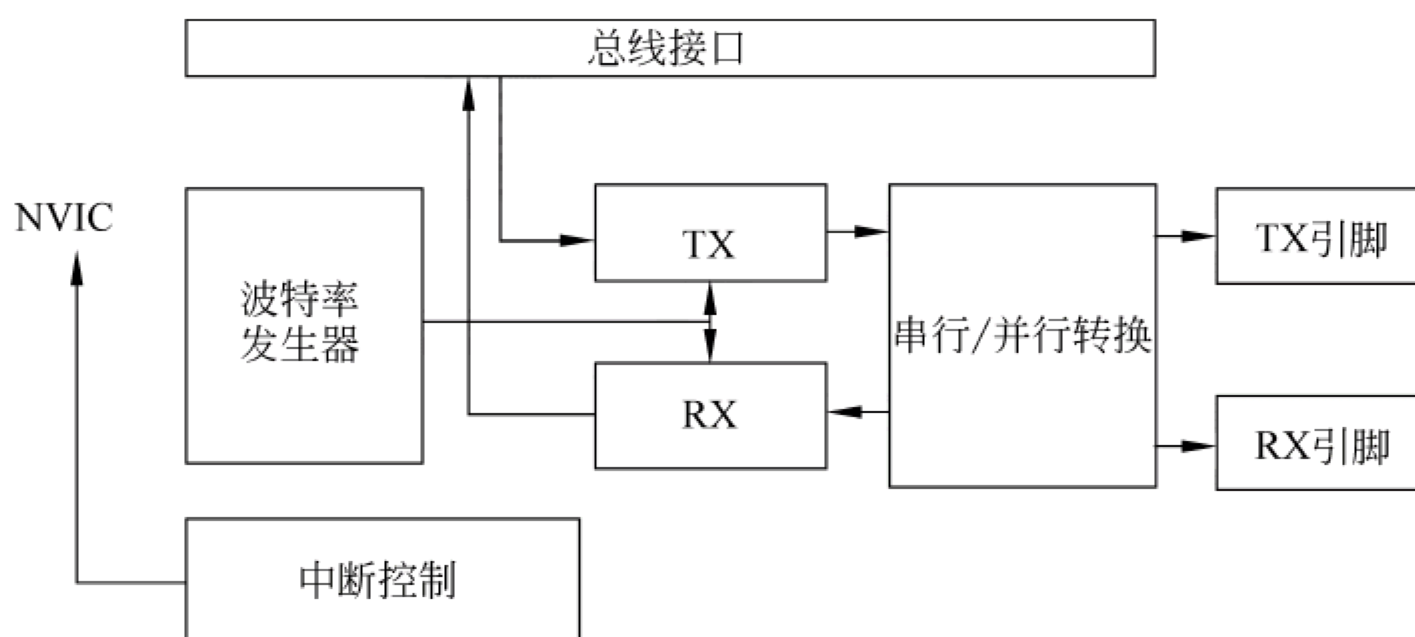


图 8-7　UART 的内部结构

8.3　STM32L152 USART 内部结构与原理

STM32L152 内部集成的通用同步异步收发器(USART)提供了一种灵活的方法与使用工业标准 NRZ 异步串行数据格式的外部设备之间进行全双工数据交换。USART 模式支持：通用全双工异步通信模式(UART)、智能卡模式(ISO7816-3，单线半双工异步模式)、通用全双工同步通信模式(USRT)、硬件流控模式(调制解调器)、IrDA 红外模式、LIN 通信模式。此外它还允许多处理器通信。为实现高速数据通信，可使用多缓冲器配置的 DMA 方式。本章主要对 USART 的异步通信 UART 进行介绍。

STM32L152 系列微控制器最多可集成 5 个串行控制器，每个串行控制器所支持的功能如图 8-8 所示。串口 1～3 是全功能串行控制器，串口 4 和串口 5 不支持同步模式、硬件流控制和智能卡模式。其控制器特点为：

(1) 可编程的波特率发生器系统，最高达 4Mb/s，采样时钟支持 8/16 倍波特率采样时钟；

(2) 可编程数据字长度(8 位或 9 位)，可配置的停止位(支持 1 或 2 个停止位)；

(3) 发送方为同步传输提供时钟；

(4) 支持全双工异步通信和单线半双工通信；

(5) 支持 LIN、SmartCard、IrDA 等多种模式；

(6) 可配置使用 DMA 的多缓冲器通信，在 SRAM 里利用集中式 DMA 缓冲接收/发送

字节；

(7) 单独控制的发送器和接收器使能位；

(8) 支持接收缓冲器满、发送缓冲器空、传输结束标志等多种检测标志，支持发送和接收校验控制；

(9) 支持多种错误检测标志，10 个带标志的中断源；

(10) 支持多处理器通信、静默模式唤醒等。

USART模式	USART1	USART2	USART3	UART4	UART5
异步模式	X	X	X	X	X
硬件流控制	X	X	X	NA	NA
多缓存通信(DMA)	X	X	X	X	X
多处理器通信	X	X	X	X	X
同步	X	X	X	NA	NA
智能卡	X	X	X	NA	NA
半双工(单线模式)	X	X	X	X	X
IrDA	X	X	X	X	X
LIN	X	X	X	X	X

图 8-8　STM32L152 USART 控制器支持的功能列表

一个全功能的串行控制器内部结构如图 8-9 所示。内部模块主要包括发送和接收单元(数据寄存器和移位寄存器)、红外编解码单元、时钟输出控制单元、发送和接收控制单元、硬件流控制单元、波特率控制单元和中断控制单元。内部涉及的寄存器包括控制寄存器 CR、状态寄存器 SR、数据寄存器 TDR、RDR，波特率因子寄存器 BRR 等。

外部引脚包括 TX、RX、IRDA_IN、IRDA_OUT、RTS、CTS 和 CK。对应的 GPIO 引脚如表 8-1 所示。

表 8-1　USART 控制器专用 I/O 引脚

USART 专用功能	UART1 I/O 引脚	UART2 I/O 引脚	UART3 I/O 引脚	UART4 I/O 引脚	UART5 I/O 引脚
TX/ IRDA_OUT	PA9、PB6	PA2、PD5	PB10、PC10、PD8	PC10	PC12
RX/ IRDA_IN	PA10、PB7	PA3、PD6	PB11、PC11、PD9	PC11	PD2
RTS	PA12	PA1、PD4	PB14、PD12		
CTS	PA11	PA0、PD3	PB13、PD11		
CK	PA8	PA4、PD7	PB12、PC12、PD10		

任何 USART 双向通信至少需要两个脚：接收数据输入(RX)和发送数据输出(TX)。当发送器被禁止时，TX 输出引脚恢复为通用 I/O 端口功能。当发送器被激活，并且不发送数据时，TX 引脚处于高电平。在单线和智能卡模式里，此 I/O 口被同时用于数据的发送和接收。

在同步模式中需要使用 CK 引脚，CK 为发送器时钟输出，用于同步传输的时钟，数据

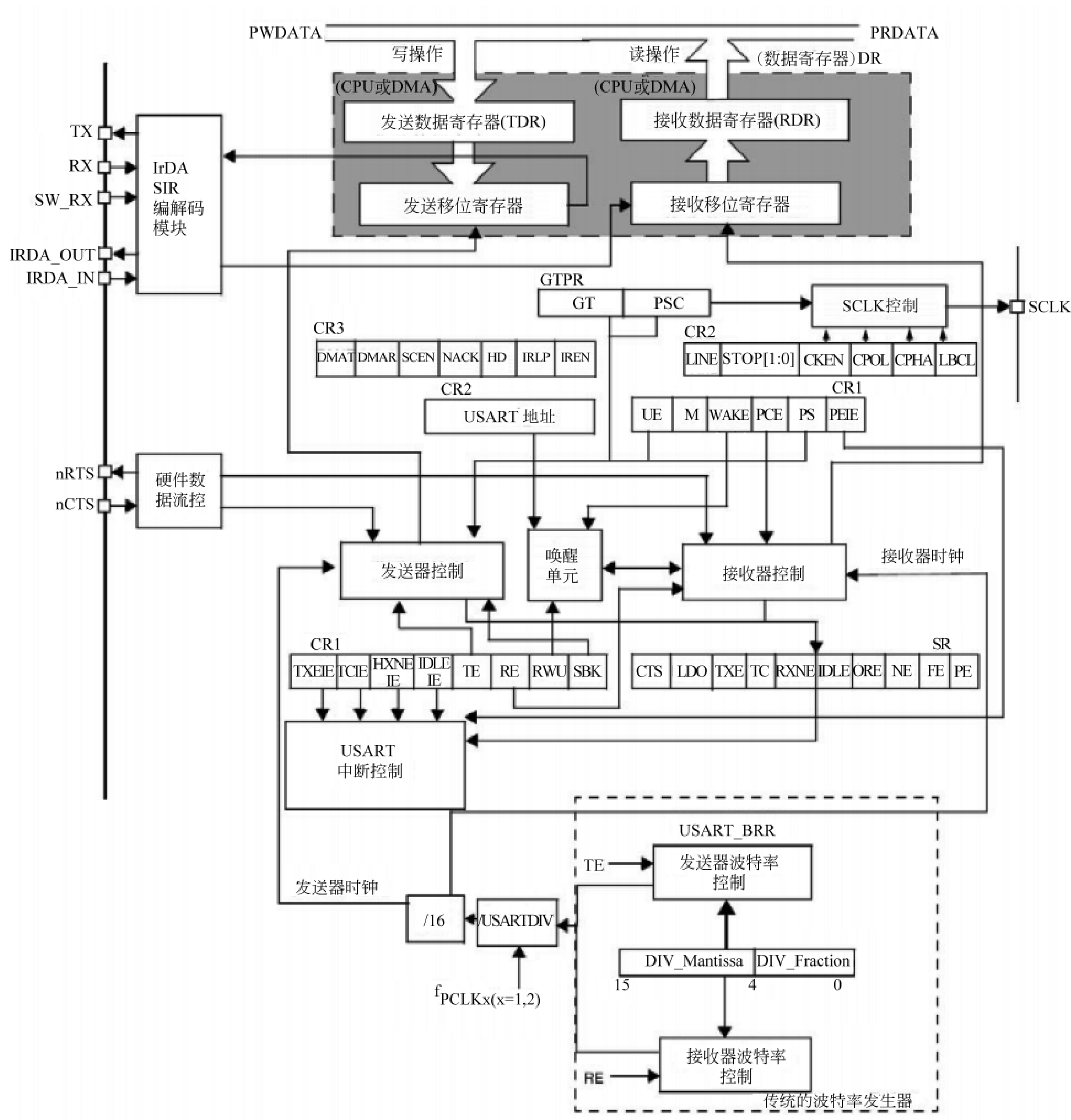


图 8-9　STM32L152 USART 控制器内部结构

可以在 RX 上同步被接收,时钟的相位和极性都是软件可编程的。在智能卡模式里,CK 可以为智能卡提供时钟。在 IrDA 模式里需要使用 IrDA_RDI(IrDA 模式下的数据输入)和 IrDA_TDO(IrDA 模式下的数据输出)引脚。在硬件流控模式中需要使用 nCTS 和 nRTS 引脚。

8.3.1　发送器

发送器发送 8 位或 9 位的数据字,字长可以通过编程 USART_CR1 寄存器中的 M 位来选择成 8 或 9 位。当发送使能位(TE)被设置时,发送移位寄存器中的数据在 TX 脚上输

出，相应的时钟脉冲在 CK 脚上输出。

1. 字符发送

在 USART 发送期间，在 TX 引脚上首先移出数据的最低有效位。对 USART_DR 寄存器的写操作实际上是对发送数据寄存器 TDR 的操作。每个字符之前都有一个低电平的起始位，之后跟着的停止位，停止位的数目可配置，USART 支持 0.5、1、1.5 和 2 个停止位，如图 8-10 所示。

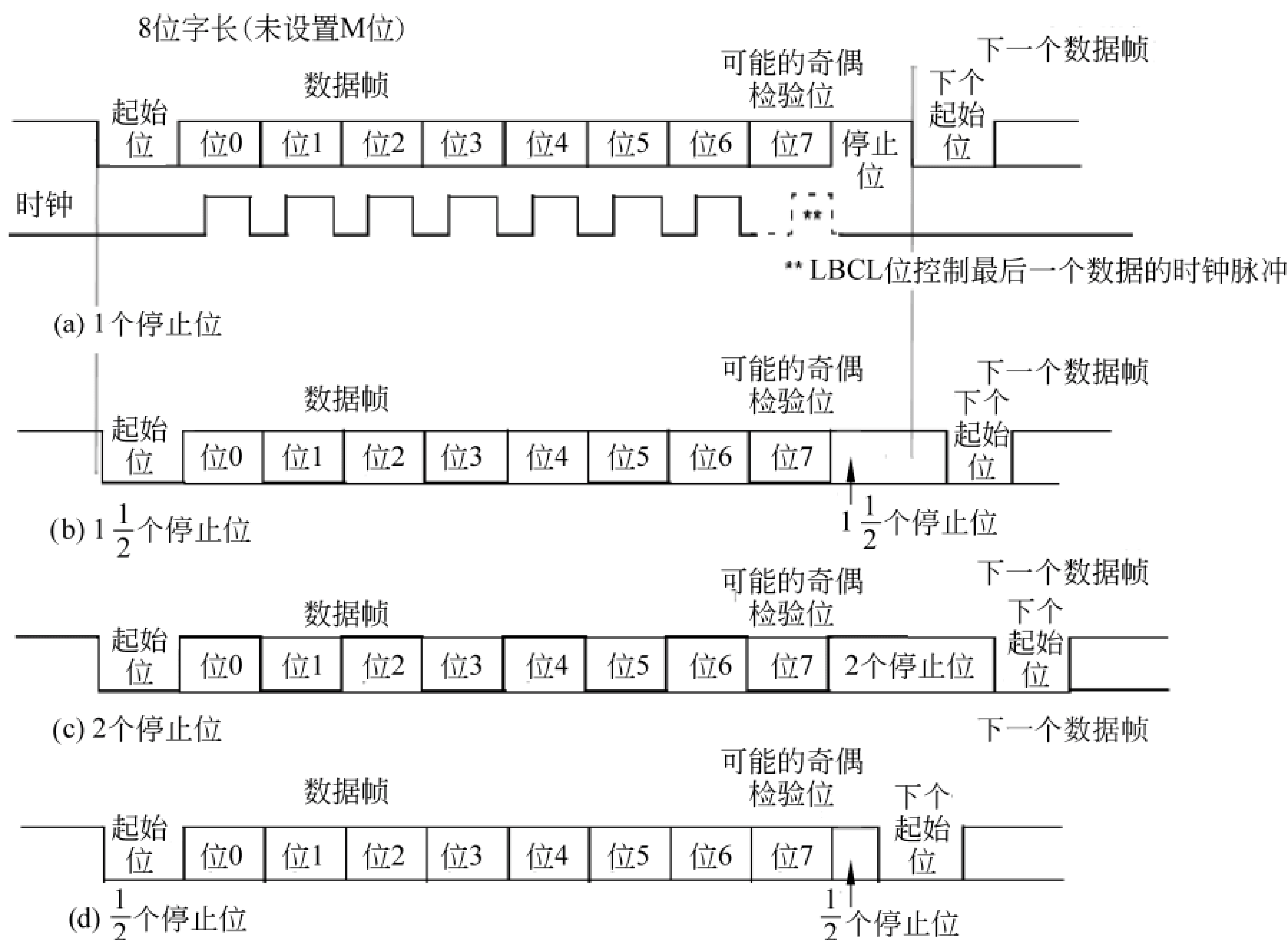


图 8-10　停止位的配置

- 1 个停止位：停止位位数的默认值。
- 2 个停止位：可用于常规 USART 模式、单线模式以及调制解调器模式。
- 0.5 个停止位：在智能卡模式下接收数据时使用。
- 1.5 个停止位：在智能卡模式下发送和接收数据时使用。

如图 8-5 所示，空闲帧包括了停止位，断开帧是 10 位低电平（M＝0）或 11 位低电平（M＝1），后跟停止位。

2. 单字节通信

发送一个字节时，需要将数据写入到数据寄存器 USART_DR，USART 控制器通过 TXE 和 TC 表示发送状态。TXE 是发送寄存器空指示标志，当发送寄存器 TDR 没有数据时置 1，清零 TXE 位是通过对数据寄存器的写操作来完成的。TXE 位由硬件来设置，它表明：

（1）数据已经从 TDR 移送到移位寄存器，数据发送已经开始；

（2）TDR 寄存器被清空；

(3) 下一个数据可以被写进 USART_DR 寄存器而不会覆盖先前的数据。

如果中断允许标志 TXEIE 位被设置为 1,则当 TXE 为 1 时将产生一个中断。如果此时 USART 正在发送数据,对 USART_DR 寄存器的写操作把数据存进 TDR 寄存器,并在当前传输结束时把该数据复制进移位寄存器。如果此时 USART 没有在发送数据,处于空闲状态,对 USART_DR 寄存器的写操作直接把数据放进移位寄存器,数据传输开始,TXE 位立即被置为 1。

当一个数据发送完成(停止位发送后)并且设置了 TXE 位为 1,TC 位被置为 1,TC 位用来表示数据传输已经完成,即当移位寄存器的最后一个数据发出以后,TC 被置 1,如果 USART_CR1 寄存器中的中断允许标志 TCIE 位被置 1 时,则会产生中断。在 USART_DR 寄存器中写入了最后一个数据字后,在关闭 USART 模块之前,必须先等待 TC=1。

发送时 TC/TXE 的状态变化如图 8-11 所示。

8.3.2 接收器

USART 可以根据 USART_CR1 寄存器的 M 位配置情况,接收 8 位或 9 位的数据字。

1. 起始位侦测

在 USART 中,如果辨认出一个特殊的采样序列,那么就认为侦测到一个起始位。该序列为 1110X0X0X0000,如图 8-12 所示。如果该序列不完整,那么接收端将退出起始位侦测并回到空闲状态(不设置标志位),等待判断下一个起始位。

在起始位的判断中,采用了 16 倍波特率时钟和过采样,如果 3 个采样点都为 0(在第 3、5、7 位的第一次采样,和在第 8、9、10 的第二次采样都为 0),则确认收到起始位,这时设置接收数据不为空标志位 RXNE=1,如果接收中断使能 RXNEIE=1,则产生中断。

如果两次 3 个采样点上仅有 2 个是 0 或一次 3 个采样点上仅有 2 个是 0(第 3、5、7 位的采样点和第 8、9、10 位的采样点),那么起始位仍然是有效的,但是会设置噪声标志位 NE。如果不能满足这个条件,则中止起始位的侦测过程,接收器会回到空闲状态(不设置标志位)。

2. 字符接收

在 USART 接收期间,数据的最低有效位首先从 RX 脚移进。USART_DR 寄存器的读操作实际上是对接收数据寄存器 RDR 的操作。接收过程中,USART 用接收数据寄存器非空标志 RXNE 表示数据是否收到,当一字符被接收到时:

- RXNE 位被置位。它表明移位寄存器的内容被转移到 RDR,即数据已经被接收并且可以被读出。
- 如果 RXNEIE 位被设置,产生中断。
- 在接收期间如果检测到各种错误,则相应的错位标志位被置 1。
- 在多缓冲器通信时(接收缓冲区是一个队列),RXNE 在每个字节接收后被置 1,并由 DMA 对数据寄存器的读操作而清零。
- 在单缓冲器模式里(接收缓冲区只能存储一个 8 位或 9 位的数据字),由软件读 USART_DR 寄存器完成对 RXNE 位清除,也可以通过对它写 0 来清除。RXNE 位必须在下一字符接收结束前被清零,以避免溢出错误。

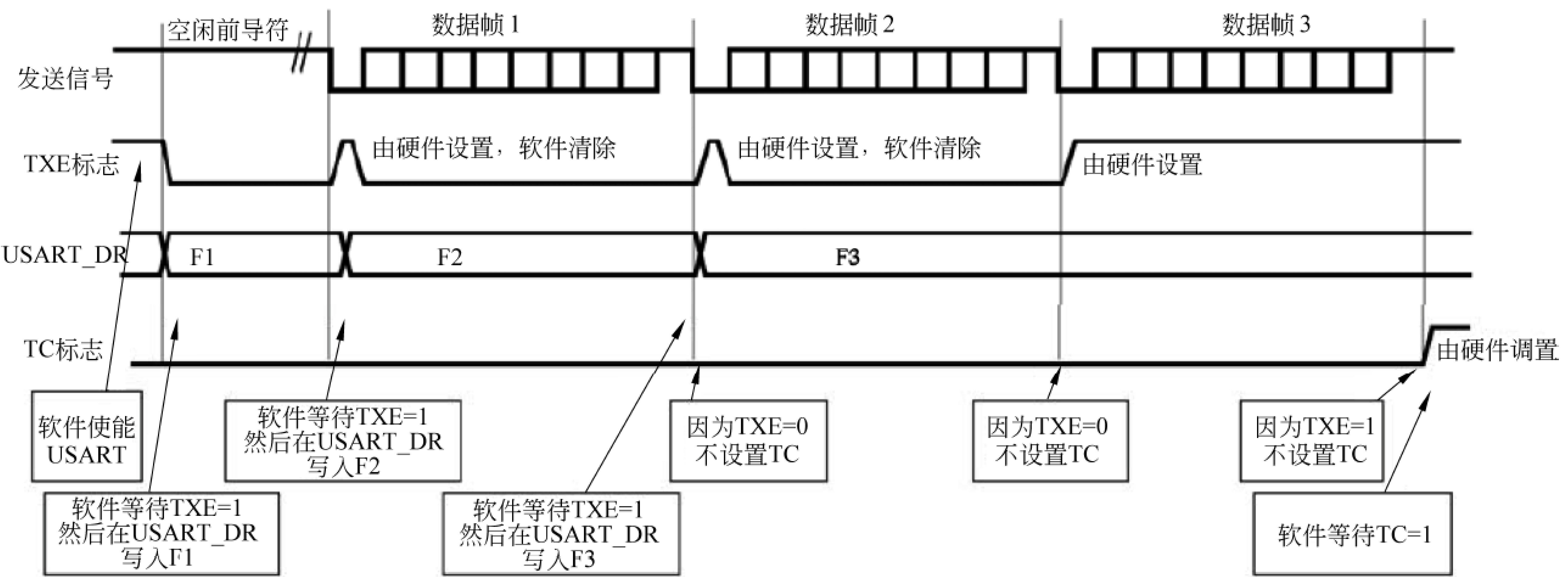


图 8-11　数据连续发送时 TXE 和 TC 的状态变化

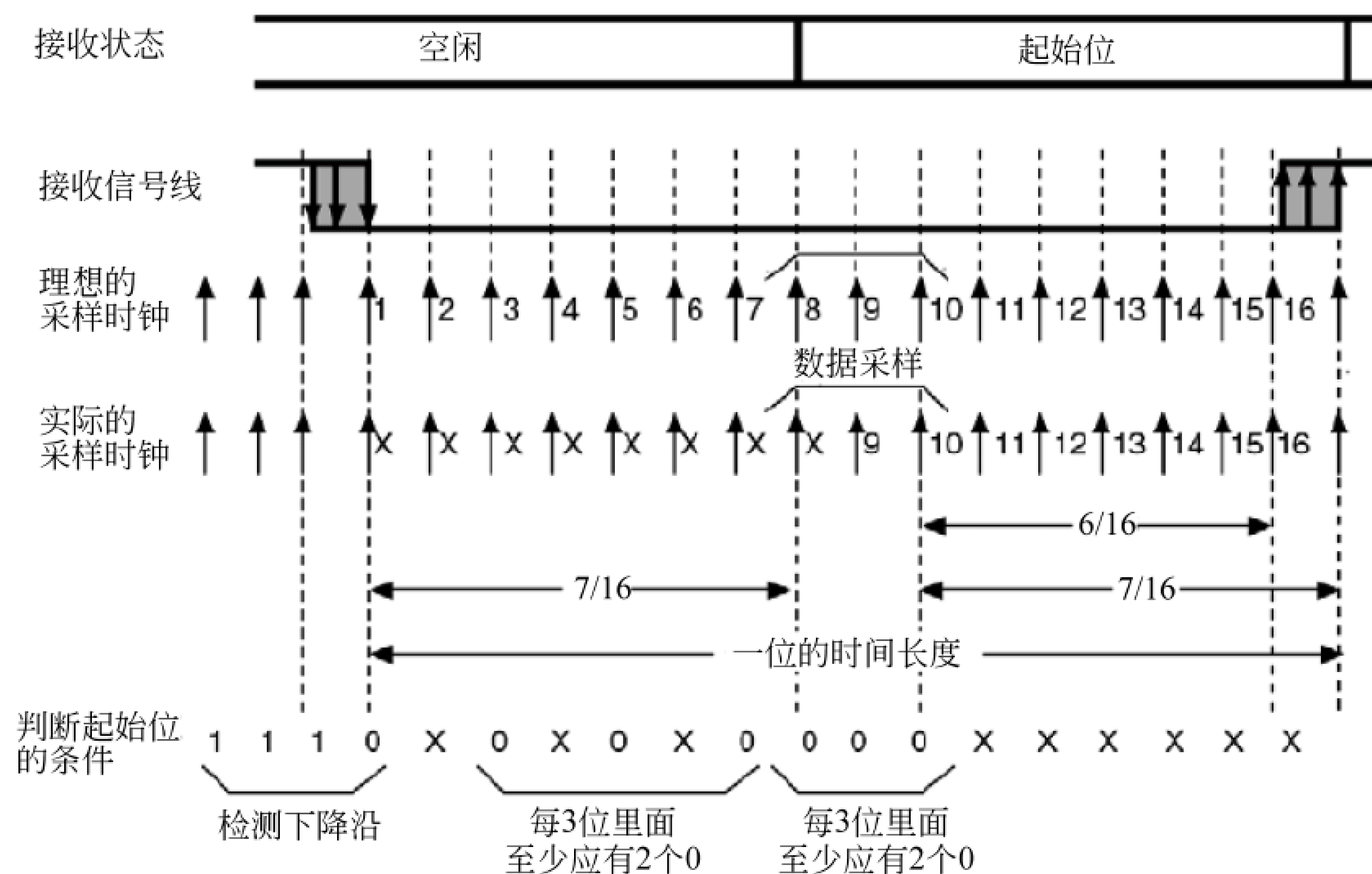


图 8-12 起始位检测

如果接收到一个断开符号，则 USART 以帧错误处理，如果收到空闲帧，其处理步骤和接收到普通数据帧一样，但如果空闲中断标志 IDLEIE 位被置 1 则将产生一个中断。

3. 接收采样时钟和过采样选择

STM32L152 USART 控制器支持 8 倍波特率和 16 倍波特率采样时钟，由控制寄存器 USART_CR1 的 OVER8 选择。异步传输的时钟精度虽然要求不高，但是也有一个容忍范围，采样时钟越高，容忍范围就越宽，但总线时钟最高 32MHz，因此数据传输的波特率必然受到限制。当 OVER8=0 时，采用 16 倍波特率采样时钟，最高波特率为 2Mb/s；当 OVER8=1 时，采用 8 倍波特率采样时钟，最高波特率可达 4Mb/s，但此时对时钟偏差的要求较高。

此外，为了提高数据抗干扰能力，可以通过控制寄存器 USART_CR3 的 ONEBIT 域开启三次采样功能。当 ONEBIT 置为 1 时，在每个有效数据位的最中间位置进行一次采样作为接收数据；当 ONEBIT 置为 0 时，在每个有效数据位的最中间进行三次采样，并对三次采样值进行投票判断，决定最终的有效数据位取值。如图 8-13 所示，16 倍波特率采样下，在时钟的 8、9、10 三个时刻进行 RX 数据采样，8 倍波特率时在时钟的 4、5、6 时刻采样。

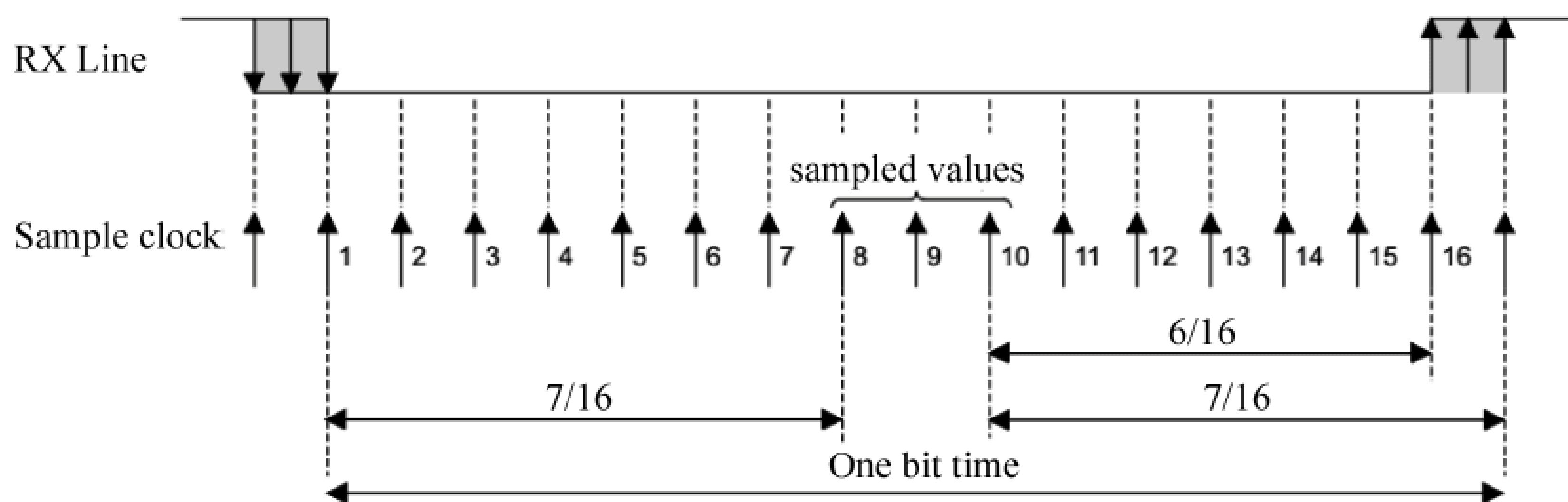


图 8-13 ONEBIT 三次采样时序

当三次采样的数据结果一致时(000 或 111),数据没有噪声,判定为有效;当三次采样的数据结果不一致时,按照少数服从多数的原则确定数据的有效电平,将数据认定为有噪声数据,并执行以下操作:

- 接收完数据,置 RXNE 位为 1,并同时设置噪声标志 NE 为 1。
- 无效数据从移位寄存器传送到 USART_DR 寄存器。
- 在单个字节通信情况下,NE 不会产生中断。然而,因为 NE 标志位和 RXNE 标志位是同时被设置的,RXNE 将产生中断。在多缓冲器通信情况下,如果已经设置了 USART_CR3 寄存器中 EIE 位,将产生一个中断。
- 先读状态寄存器 USART_SR,再读数据寄存器 USART_DR,将清除 NE 标志位。

图 8-14 是在不同采样频率和是否启用三次采样的情况下串口波特率能容忍的最大时钟误差,由图 8-14 可见,在 16 倍波特率相对 8 倍波特率时钟偏差的容忍度更大;三次采样比一次采样时钟偏差的容忍度更大。

M bit	OVER8 bit = 0		OVER8 bit = 1	
	ONEBIT=0	ONEBIT=1	ONEBIT=0	ONEBIT=1
0	3.75%	4.375%	2.50%	3.75%
1	3.41%	3.97%	2.27%	3.41%

图 8-14　串口能容忍的最大时钟误差

4. 接收中的错误

1) 溢出错误

如果 RXNE 还没有被复位,又接收到一个字符,则发生溢出错误。数据只有当 RXNE 位被清零后才能从移位寄存器转移到 RDR 寄存器。RXNE 标记是接收到每个字节后被置 1 的。如果下一个数据已被收到或先前 DMA 请求还没被服务时,RXNE 标志仍是置 1 的,溢出错误产生。当溢出错误产生时:

- 溢出错误标志位 ORE 被置 1;
- 接收数据寄存器 RDR 内容不会丢失,读 USART_DR 寄存器仍能得到先前的数据;
- 移位寄存器中以前的内容将被覆盖,随后接收到的数据都将丢失;
- 如果 RXNEIE 位被设置或 EIE 和 DMAR 位都被设置,中断产生;
- 顺序执行对 USART_SR 和 USART_DR 寄存器的读操作,可复位 ORE 位。

2) 噪音错误

当 ONEBIT 被置 1 启用三次采样时,若收到的三个采样值不一致,则产生噪声错误。噪声错误发生时,数据仍然被报错到 RDR 寄存器,但会置 NE 标志位,由用户决定是否使用该数据。

3) 帧错误

由于没有同步上或大量噪音的原因,停止位没有在预期的时间上接和收识别出来时产生帧错误,当帧错误被检测到时:

- FE 位被硬件置 1;

• 无效数据从移位寄存器传送到 USART_DR 寄存器，RXNE 置为 1；
• 在单字节通信时，FE 没有中断产生，但由于 RXNE 位置 1 产生中断，可通过 RXNE 中断服务对 FE 进行判断。在多缓冲器通信情况下，如果 USART_CR3 寄存器中 EIE 位被置位的话，将产生中断；
• 顺序执行对 USART_SR 和 USART_DR 寄存器的读操作，可复位 FE 位。

8.3.3 校验控制

设置 USART_CR1 寄存器上的 PCE 位，可以使能奇偶控制（发送时生成一个奇偶位，接收时进行奇偶校验）。根据 M 位定义的帧长度和是否启用校验，USART 的帧格式如表 8-2 所示。

表 8-2 启用/不启用校验时的数据帧格式

M 位	PCE 位	USART 帧
0	0	|起始位 | 8 位数据 | 停止位 |
0	1	|起始位 | 7 位数据 | 奇偶检验位 | 停止位 |
1	0	|起始位 | 9 位数据 | 停止位 |
1	1	|起始位 | 8 位数据 | 奇偶检验位 | 停止位 |

偶校验指的是校验位使得一帧中的 7 或 8 个数据位以及校验位中 1 的个数为偶数。奇校验指的是校验位使得一帧中的 7 或 8 个数据位以及校验位中 1 的个数为奇数。例如数据为 00110101，有 4 个 1，如果选择偶校验，校验位将是 0。如果选择奇校验，校验位将是 1。

如果启用校验，写进数据寄存器 TDR 的数据的最后一位被校验位替换后发送出去，如果奇偶校验失败，状态寄存器 USART_SR 寄存器中的 PE 标志被置 1，如果控制寄存器 USART_CR1 寄存器的 PEIE 被置 1，则产生一个中断。

8.3.4 硬件流控制

利用 nCTS 输入和 nRTS 输出可以控制两个设备间的串行数据流。通过将控制寄存器 UASRT_CR3 中的 RTSE 和 CTSE 置 1 可以分别独立地使能 RTS 和 CTS 流控制。

1. RTS 流控制

如图 8-15 所示，如果 RTS 流控制被使能，只要 USART 接收器准备好接收新的数据，nRTS 就变成有效（低电平）。当接收寄存器内有数据到达时，nRTS 被置无效（高电平），由此表明希望在当前帧结束时停止数据传输。

2. CTS 流控制

如图 8-16 所示，如果 CTS 流控制被使能，发送器在发送下一帧前检查 nCTS 输入。如果 nCTS 有效（低电平），则下一个数据被发送，否则下一帧数据不被发出去。若 nCTS 在传

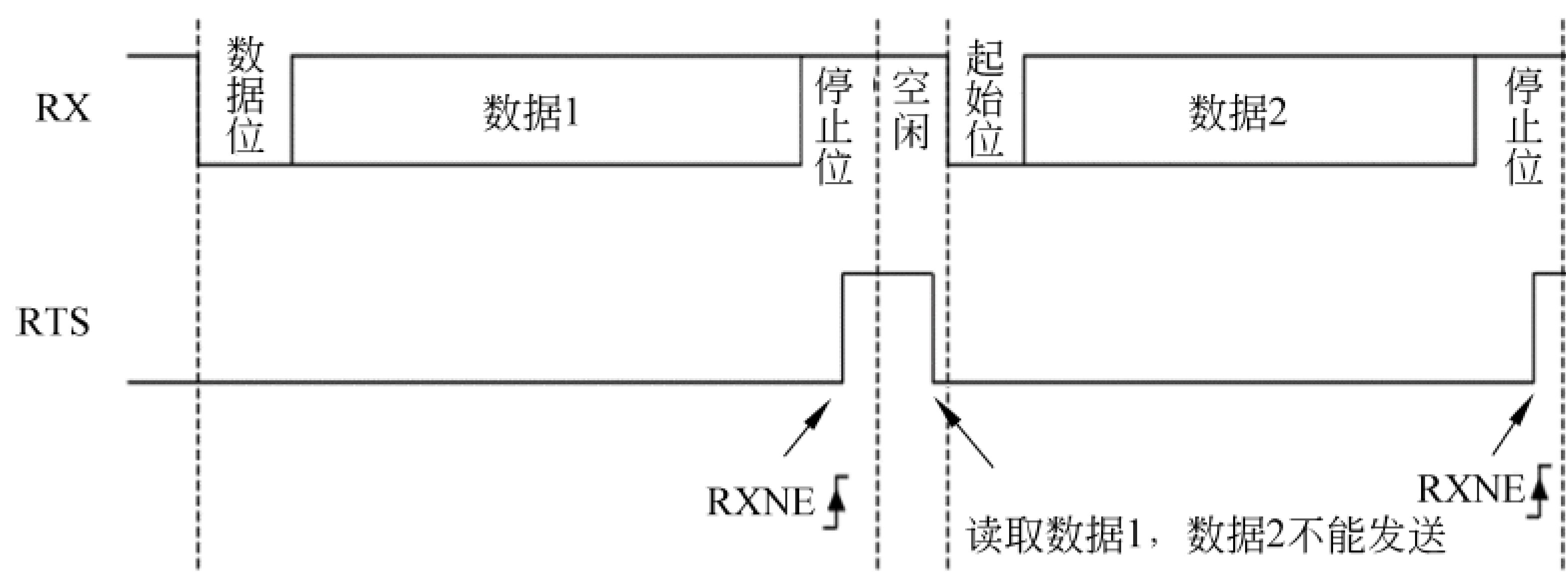


图 8-15　RTS 流控制时序

输期间被变成无效(高电平),当前的传输完成后停止发送。在启用 CTS 流控制时,只要 nCTS 线的输入状态发生变化,硬件就自动设置 CTSIF 状态位为 1。如果设置了 USART_CT3 寄存器的 CTSIE 位,则产生中断。

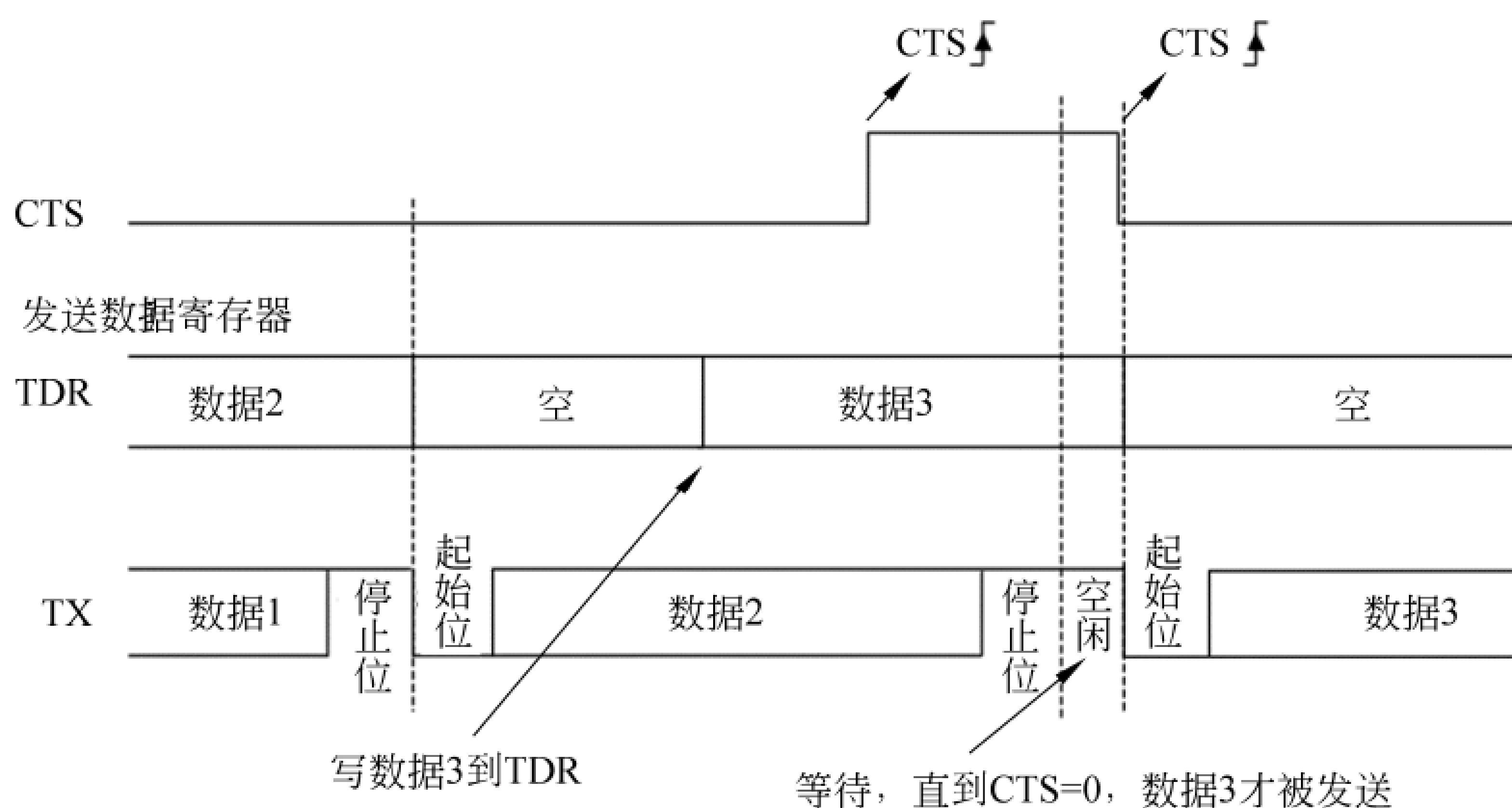


图 8-16　CTS 流控制时序

8.3.5　USART 中断请求

如表 8-3 所示,USART 控制器内部有多达 10 个中断事件,每个事件都有一个中断使能位用于决定是否可以向 NVIC 发起中断请求。各种中断事件被连接到同一个中断向量,其连接关系如图 8-17 所示。

表 8-3　USART 中断源

中断事件	事件标志	使能位
发送数据寄存器空	TXE	TXEIE
CTS 标志	CTS	CTSIE

续表

中 断 事 件	事 件 标 志	使　能　位
发送完成	TC	TCIE
接收数据就绪可读	RXNE	RXNEIE
检测到数据溢出	ORE	RXNEIE
检测到空闲线路	IDLE	IDLEIE
奇偶检验错	PE	PEIE
断开标志	LBD	LBDIE
DMA 多缓存区通信下的噪声标志、溢出错误和帧错误	NE 或 ORT 或 FE	EIE

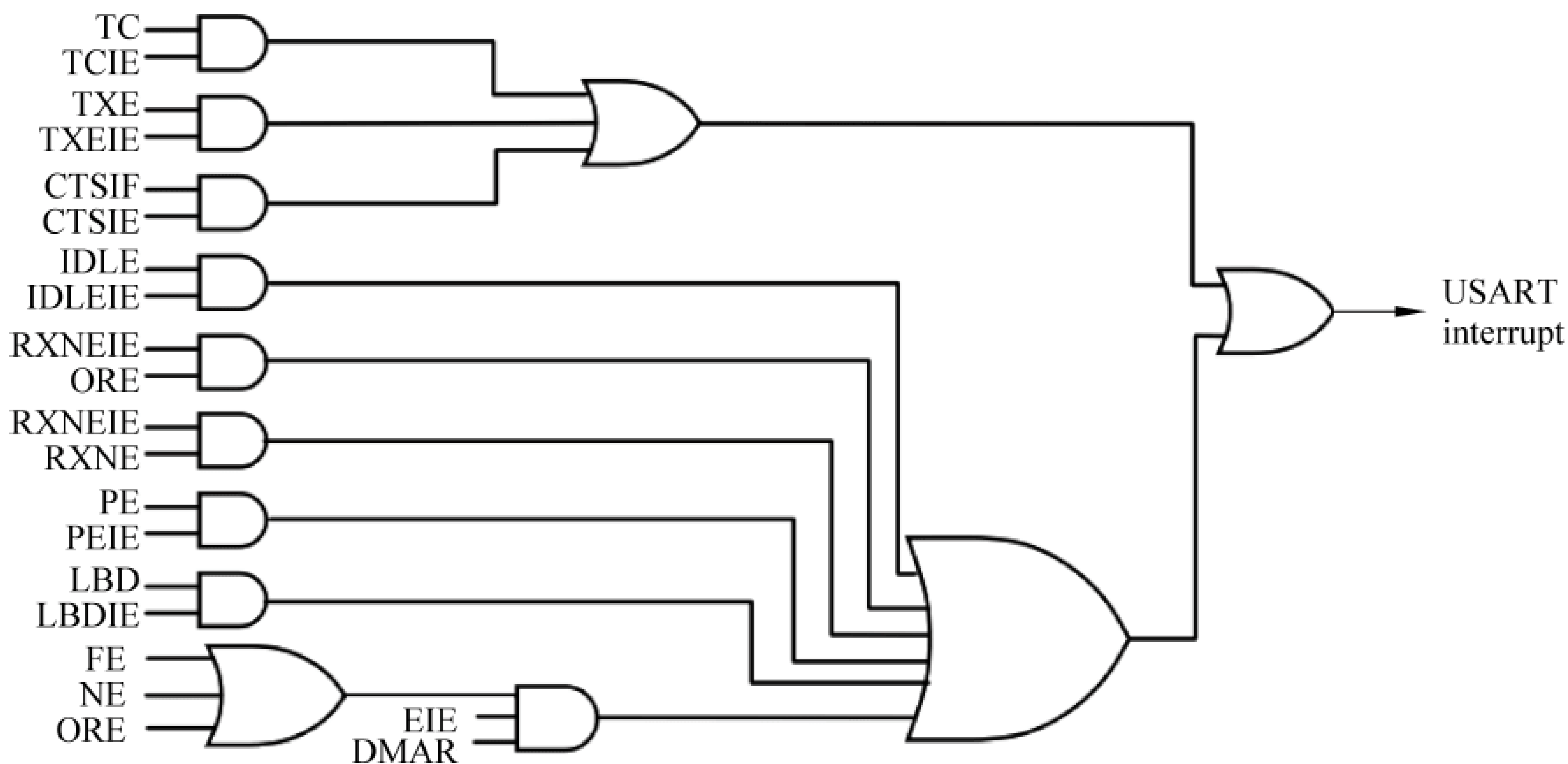


图 8-17　USART 内部中断源连接关系

8.4　USART 寄存器

USART 控制器的寄存器如表 8-4 所示。

表 8-4　USART 控制器寄存器

寄存器名称	地址偏移量	功　　能	复　位　值
状态寄存器(USART_SR)	0x00	状态标志位	0x00C0 0000
数据寄存器(USART_DR)	0x04	发送和接收数据	0xXXXX XXXX
波特率寄存器(USART_BRR)	0x08	配置传输波特率因子	0x0000 0000
控制寄存器(USART_CR1)	0x0C	配置发送、接收及中断	0x0000 0000

续表

寄存器名称	地址偏移量	功　　能	复　位　值
控制寄存器(USART_CR2)	0x10	配置停止位、地址、同步时钟及 LIN	0x0000 0000
控制寄存器(USART_CR3)	0x14	配置采样、流控制、红外和 DMA	0x0000 0000
保护时间和分频寄存器(USART_GTPR)	0x18	配置智能卡和红外模式的保护时间和频率	0x0000 0000

1. 状态寄存器 USART_SR

状态寄存器 USART_SR 如图 8-18 所示，其有效域定义如下：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						CTS	LBD	TXE	TC	RXNE	IDLE	ORE	NF	FE	PE
						rc_w0	rc_w0	r	rc_w0	rc_w0	r	r	r	r	r

图 8-18　状态寄存器

CTS：CTS 标志位，该位为 0 表示 nCTS 状态线上没有变化，为 1 表示 nCTS 状态线上发生变化。如果设置了 CTSE 位，当 nCTS 输入变化状态时，该位被硬件置 1。由软件将其清 0。如果 USART_CR3 中的 CTSIE 为 1，则产生中断。

LBD：LIN 断开检测(LIN Break Detect)，当探测到 LIN 断开时，该位由硬件置 1，由软件清 0。如果 USART_CR3 中的 LBDIE=1，则产生中断。

TXE：发送寄存器空标志位，当 TDR 寄存器中的数据被硬件转移到移位寄存器的时候，该位被硬件置 1。如果 USART_CR1 寄存器中的 TXEIE 为 1，则产生中断。对 USART_DR 的写操作，将该位清 0。该位为 0 表示数据还没有被转移到移位寄存器，为 1 表示数据已经被转移到移位寄存器。

TC：发送完成标志位，该位为 0 表示发送还未完成，1 表示发送完成。当包含有数据的一帧发送完成后，并且 TXE=1 时，由硬件将该位置 1。如果 USART_CR1 中的 TCIE 为 1，则产生中断。由软件序列清除该位(先读 USART_SR，然后写 USART_DR)。TC 位也可以通过写入 0 来清除(多缓存通信中使用)。

RXNE：读数据寄存器非空标志位 RXNE，该位为 0 表示数据没有收到，1 表示收到数据，可以读出。当 RDR 移位寄存器中的数据被转移到 USART_DR 寄存器中，该位被硬件置位。如果 USART_CR1 寄存器中的 RXNEIE 为 1，则产生中断。对 USART_DR 的读操作可以将该位清零。RXNE 位也可以通过写入 0 来清除(在多缓存通信中使用)。

IDLE：总线空闲标志位，该位为 0 表示没有检测到空闲总线，为 1 表示检测到空闲总线。当检测到总线空闲时，该位被硬件置 1。如果 USART_CR1 中的 IDLEIE 为 1，则产生中断。由软件序列清除该位(先读 USART_SR，然后读 USART_DR)。

ORE：过载错误标志位，该位为 0 表示没有过载错误，为 1 表示检测到过载错误。当 RXNE 仍然是 1 的时候，当前被接收在移位寄存器中的数据，需要传送至 RDR 寄存器时，

硬件将该位置 1。如果 USART_CR1 中的 RXNEIE 为 1,则产生中断。由软件序列将其清零(先读 USART_SR,然后读 USART_CR)。该位被置位时,RDR 寄存器中的值不会丢失,但是移位寄存器中的数据会被覆盖。如果设置了 EIE 位,在多缓冲器通信模式下,ORE 标志置位会产生中断。

NE:噪声错误标志,该位为 0 表示没有检测到噪声,为 1 表示检测到噪声。在接收到的帧检测到噪音时,由硬件对该位置 1。由软件序列对其清 0(先读 USART_SR,再读 USART_DR)。该位不会产生中断,但因为它和 RXNE 一起出现,硬件会在设置 RXNE 标志时产生中断。在多缓冲区通信模式下,如果设置了 EIE 位,则设置 NE 标志时会产生中断。

FE:帧错误标志位 FE,该位为 0 表示没有检测到帧错误,为 1 表示检测到帧错误或者 break 符。当检测到同步错位、过多的噪声或者检测到断开符,该位被硬件置 1。由软件序列将其清零(先读 USART_SR,再读 USART_DR)。该位不会产生中断,但因为它和 RXNE 一起出现,硬件会在设置 RXNE 标志时产生中断。如果当前传输的数据既产生了帧错误,又产生了过载错误,硬件还是会继续该数据的传输,并且只设置 ORE 标志位。在多缓冲区通信模式下,如果设置了 EIE 位,则设置 FE 标志时会产生中断。

PE:校验错误标志位,该位为 0 表示没有奇偶校验错误,为 1 表示检测到奇偶校验错误。在接收模式下,如果出现奇偶校验错误,硬件对该位置 1。由软件序列对其清 0(依次读 USART_SR 和 USART_DR)。如果 USART_CR1 中的 PEIE 为 1,则产生中断。

2. 数据寄存器 USART_DR

数据寄存器如图 8-19 所示,由两个寄存器组成的,一个给发送用(TDR),一个给接收用(RDR),该寄存器兼具读和写的功能。TDR 寄存器提供了内部总线和输出移位寄存器之间的并行接口,RDR 寄存器提供了输入移位寄存器和内部总线之间的并行接口。



图 8-19 数据寄存器

数据寄存器的有效域为 DR[8:0],表示数据值,包含了发送或接收的数据。当使能校验位(USART_CR1 中 PCE 位被置位)进行发送时,写到 MSB 的值(根据数据的长度不同,MSB 是第 7 位或者第 8 位)会被后来的校验位取代。当使能校验位进行接收时,读到的 MSB 位是接收到的校验位。

3. 波特率配置寄存器 USART_BRR

波特率配置寄存器 USART_BRR 如图 8-20 所示,其有效域定义如下:

DIV_Mantissa[11:0]:USARTDIV 的整数部分,定义了 USART 分频器除法因子(USARTDIV)的整数部分。

DIV_Fraction[3:0]:USARTDIV 的小数部分,定义了 USART 分频器除法因子(USARTDIV)的小数部分。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DIV_Mantissa[11:0]												DIV_Fraction[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 8-20　波特率寄存器

4. 控制寄存器 USART_CR1

控制寄存器 USART_CR1 如图 8-21 所示，其有效域定义如下：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OVER8	Reserved	UE	M	WAKE	PCE	PS	PEIE	TXEIE	TCIE	RXNEIE	IDLEIE	TE	RE	RWU	SBK
rw	Res.	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 8-21　控制寄存器 1

OVER8：采样时钟选择位，该位为 1 表示采用发送/接收时钟的 8 分频作为波特率，为 0 表示采用发送/接收时钟的 16 分频作为波特率。

UE：USART 使能（USART enable），该位被置 0，表示禁用 USART 模块，在当前字节传输完成后 USART 的分频器和输出停止工作。置 1 表示 USART 模块使能。

M：字长，该位定义了数据字的长度，由软件对其置 1 和清 0，0 表示 8 个数据位，1 表示 9 个数据位，在数据传输过程中（发送或者接收时），不能修改这个位。

WAKE：静默模式唤醒方法，这位决定将 USART 从静默模式唤醒的方法，0 表示被空闲总线唤醒，1 表示被地址标记唤醒，由软件对该位置 1 和清 0。

PCE：检验控制使能（Parity control enable），0 表示禁止校验控制，1 表示使能校验控制。用该位选择是否进行硬件校验控制（对于发送来说就是校验位的产生；对于接收来说就是校验位的检测）。当使能了该位，在发送数据的最高位（如果 M=1，最高位就是第 9 位；如果 M=0，最高位就是第 8 位）插入校验位；对接收到的数据检查其校验位。一旦设置了该位，当前字节传输完成后，校验控制才生效。

PS：校验选择（Parity selection），当校验控制 PCE 置 1 后，PS 用于选择采用偶校验还是奇校验。0 表示偶校验，1 表示奇校验。

PEIE：PE 中断使能（PE interrupt enable），0 表示禁止产生中断，1 表示当 USART_SR 中的 PE 为 1 时，产生 USART 中断。

TXEIE：发送缓冲区空中断使能（TXE interrupt enable），0 表示禁止产生中断，1 表示当 USART_SR 中的 TXE 为 1 时，产生 USART 中断。

TCIE：发送完成中断使能（Transmission complete interrupt enable），0 表示禁止产生中断，1 表示当 USART_SR 中的 TC 为 1 时，产生 USART 中断。

RXNEIE：接收缓冲区非空中断使能（RXNE interrupt enable），0 表示禁止产生中断，1 表示当 USART_SR 中的 ORE 或者 RXNE 为 1 时，产生 USART 中断。

IDLEIE：IDLE 中断使能（IDLE interrupt enable），0 表示禁止产生中断，1 表示当 USART_SR 中的 IDLE 为 1 时，产生 USART 中断。

TE：发送使能（Transmitter enable），0 表示禁止发送，1 表示使能发送，当 TE 被设置后，在真正发送开始之前，有一个比特时间的延迟。

RE：接收使能（Receiver enable），0 表示禁止接收，1 表示使能接收，并开始搜寻 RX 引脚上的起始位。

RWU：接收唤醒（Receiver wakeup），该位用来决定是否把 USART 置于静默模式，0 表示处于正常工作模式，1 表示接收器处于静默模式，当唤醒序列到来时，硬件将其清零。

SBK：发送断开帧（Send break），使用该位来发送断开字符。0 表示没有发送断开字符，1 表示将要发送断开字符。

5. 控制寄存器 USART_CR2

控制寄存器 USART_CR2 如图 8-22 所示，其有效域定义如下：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Res.	LINEN	STOP[1:0]		CLKEN	CPOL	CPHA	LBCL	Res.	LBDIE	LBDL	Res.	ADD[3:0]			
	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw	rw	rw	rw

图 8-22　控制寄存器 2

LINEN：LIN 模式使能（LIN mode enable），0 表示禁止 LIN 模式，1 表示开启。

STOP：停止位（STOP bits），这 2 位用来设置停止位的位数，00 表示 1 个停止位，01 表示 0.5 个停止位，10 表示 2 个停止位，11 表示 1.5 个停止位，同步和异步串行模式智能使用 1 个或 2 个停止位。

CLKEN：时钟使能（Clock enable），该位用来使能 CK 引脚，0 表示禁止 CK 引脚，1 表示使能 CK 引脚。UART4 和 UART5 上不存在这一位。

CPOL：时钟极性（Clock polarity），在同步模式下，可以用该位选择 SLCK 引脚上时钟输出的极性。和 CPHA 位一起配合来产生需要的时钟/数据的采样关系，0 表示总线空闲时 CK 引脚上保持低电平，1 表示总线空闲时 CK 引脚上保持高电平。

CPHA：时钟相位（Clock phase），在同步模式下，可以用该位选择 SLCK 引脚上时钟输出的相位。0 表示在时钟的第一个边沿进行数据捕获，1 表示在时钟的第二个边沿进行数据捕获。

LBCL：最后一个字节的时钟控制，在同步模式下，使用该位来控制是否在 CK 引脚上输出最后发送的那个数据字节（MSB）对应的时钟脉冲，0 表示不输出，1 表示输出。UART4 和 UART5 没有 CPOL、CPHA 和 LBCL 域。

LBDIE：LIN 断开符检测中断使能（LIN break detection interrupt enable），断开符中断屏蔽（使用断开分隔符来检测断开符），0 表示禁止中断，1 表示 USART_SR 寄存器中的 LBD 为 1 就产生中断。

LBDL：LIN 断开符检测长度（LIN break detection length），该位用来选择是 11 位还

是 10 位的断开符检测，0 表示 10 位，1 表示 11 位。

ADD[3：0]：本设备的 USART 节点地址，在多处理器通信下的静默模式中使用的，使用地址标记来唤醒某个 USART 设备。

6. 控制寄存器 USART_CR3

控制寄存器 USART_CR3 如图 8-23 所示，其有效域定义如下：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Reserved				ONEBIT	CTSIE	CTSE	RTSE	DMAT	DMAR	SCEN	NACK	HDSEL	IRLP	IREN	EIE
				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 8-23　控制寄存器 3

ONEBIT：过采样使能位，为 1 表示只进行一次采样，为 0 表示进行三次采样。

CTSIE：CTS 中断使能（CTS interrupt enable），0 表示禁止中断，1 表示 USART_SR 寄存器中的 CTS 为 1 时产生中断。UART4 和 UART5 上不存在这一位。

CTSE：CTS 使能（CTS enable），0 表示禁止 CTS 硬件流控制，1 表示 CTS 模式使能，只有 nCTS 输入信号有效(拉成低电平)时才能发送数据。如果在数据传输的过程中，nCTS 信号变成无效，那么发完这个数据后，传输就停止下来。如果当 nCTS 为无效时，往数据寄存器里写数据，则要等到 nCTS 有效时才会发送这个数据。UART4 和 UART5 上不存在这一位。

RTSE：RTS 使能（RTS enable），0 表示禁止 RTS 硬件流控制，1 表示 RTS 使能，只有接收缓冲区内有空余的空间时才请求下一个数据。当前数据发送完成后，发送操作就需要暂停下来。如果可以接收数据了，将 nRTS 输出置为有效(拉至低电平)。UART4 和 UART5 上不存在这一位。

DMAT：发送 DMA 使能(DMA enable Trasmit)，0 表示禁止发送时的 DMA，1 表示使能发送时的 DMA；UART4 和 UART5 上不存在这一位。

DMAR：DMA 使能接收（DMA enable receiver），0 表示禁止接收时 DMA，1 表示使能接收时的 DMA；UART4 和 UART5 上不存在这一位。

SCEN：智能卡模式使能（Smartcard mode enable），该位用来使能智能卡模式，0 表示禁止，1 表示使能。

NACK：智能卡 NACK 使能（Smartcard NACK enable），0 表示校验错误时，不发送 NACK，1 表示校验错误出现时发送 NACK。

HDSEL：半双工选择（Half-duplex selection），选择单线半双工模式，1 表示选择半双工模式。

IRLP：红外低功耗（IrDA low-power），该位用来选择普通模式还是低功耗红外模式，1 表示低功耗模式。

IREN：红外模式使能（IrDA mode enable），1 表示使能红外模式。

EIE：错误中断使能（Error interrupt enable），在多缓冲区通信模式下，当有帧错误、过载或者噪声错误时(USART_SR 中的 FE=1，或者 ORE=1，或者 NE=1)产生中断的使能

位。0 表示禁止中断，1 表示只要 USART_CR3 中的 DMAR＝1，并且 USART_SR 中的 FE＝1，或者 ORE＝1，或者 NE＝1，则产生中断。

8.5 USART 数据传输配置

8.5.1 波特率计算

发送和接收方的波特率必须要配置为一致的数值，常用的波特率位 1200、2400、4800、9600、19200、57600、115200 等，其计算公式为：

$$\text{TX/RX baud}=\frac{f_{CK}}{8\times(2-\text{OVER8})\times\text{USARTDIV}}$$

其中 f_{CK} 为总线时钟，OVER8 为 16 倍波特率/8 倍波特率选择，USARTDIV 为配置给波特率寄存器 USART_BRR 的值，由 12 位整数和 4 位小数表示。

OVER8＝0 时，USARTDIV ＝ DIV_Mantissa[11：0] ＋ DIV_Fraction[3：0]/16；

OVER8＝1 时，USARTDIV ＝ DIV_Mantissa[11：0] ＋ DIV_Fraction[2：0]/8。

例如，f_{CK}＝32MHz，采样时钟为 8 倍波特率，要配置波特率为 57600，则 USART_BRR 寄存器的值计算过程为：

$$\text{USARTDIV}=f_{CK}/\text{baud}/8 = 32\text{M}/57600/8 = 69.44$$

因此，DIV_Mantissa ＝ 69，DIV_Fraction＝0.44×8 ＝ 3.52，取值为 4。实际波特率与所需要的波特率存在偏差。波特率在 16MHz 和 32MHz 下的配置值如图 8-24 和图 8-25 所示。

16倍采样时钟（OVER8=0）							
波特率		f_{PCLK}=16MHz			f_{PCLK}=32MHz		
序号	需求波特率/kb/s	实际波特率/kb/s	波特率寄存器值	误差%	实际波特率/kb/s	波特率寄存器值	误差/%
1	1.2	1.2	833.3125	0	1.2	1666.6875	0
2	2.4	2.4	416.6875	0	2.4	833.3125	0
3	9.6	9.598	104.1875	0.02	9.601	208.3125	0.01
4	19.2	19.208	52.0625	0.04	19.196	104.1875	0.02
5	38.4	38.369	26.0625	0.08	38.415	52.0625	0.04
6	57.6	57.554	17.375	0.08	57.554	34.75	0.08
7	115.2	115.108	8.6875	0.08	115.108	17.375	0.08
8	230.4	231.884	4.3125	0.64	230.216	8.6875	0.08
9	460.8	457.143	2.1875	0.79	463.768	4.3125	0.64
10	921.6	941.176	1.0625	2.12	914.286	2.1875	0.79
11	2	NA	NA	NA	2000	1	0
12	4	NA	NA	NA	NA	NA	NA

图 8-24　16 倍波特率下的 USARTDIV 取值

8倍采样时钟（OVER8=1）							
波特率		fPCLK=16MHz			fPCLK=32MHz		
序号	需求波特率/kb/s	实际波特率/kb/s	波特率寄存器值	误差/%	实际波特率/kb/s	波特率寄存器值	误差/%
1	1.2	1.2	1666.625	0	1.2	3333.375	0
2	2.4	2.4	833.375	0	2.4	1666.625	0
3	9.6	9.598	208.375	0.02	9.601	416.625	0.01
4	19.2	19.208	104.125	0.04	19.196	208.375	0.02
5	38.4	38.369	52.125	0.08	38.415	104.125	0.04
6	57.6	57.554	34.75	0.08	57.554	69.5	0.08
7	115.2	115.108	17.375	0.08	115.108	34.75	0.08
8	230.4	231.884	8.625	0.64	230.216	17.375	0.08
9	460.8	457.143	4.375	0.79	463.768	8.625	0.64
10	921.6	941.176	2.125	2.12	914.286	4.375	0.79
11	2	2000	1	0	2000	2	0
12	4	NA	NA	NA	4000	1	0

图 8-25　8 倍波特率下的 USARTDIV 取值

8.5.2　异步双向通信模式配置

1）发送配置

- 通过在 USART_CR1 寄存器上置位 UE 位来激活 USART；
- 编程 USART_CR1 的 M 位来定义字长；
- 在 USART_CR2 中编程停止位 STOP 的位数；
- 利用 USART_BRR 寄存器选择要求的波特率；
- 如果采用多缓冲器通信，配置 USART_CR3 中的 DMA 使能位(DMAT)，按多缓冲器通信中的描述配置 DMA 寄存器；
- 设置 USART_CR1 中的 TE 位，发送一个空闲帧作为第一次数据发送；
- 把要发送的数据写进 USART_DR 寄存器(此动作清除 TXE 位)，重复此步骤发送其他数据；
- 在 USART_DR 寄存器中写入最后一个数据字后，要等待 TC=1，它表示最后一个数据帧的传输结束。当需要关闭 USART 或需要进入停机模式之前，需要确认传输结束，避免破坏最后一次传输。

2）接收配置

- 将 USART_CR1 寄存器的 UE 置 1 来激活 USART；

- 编程 USART_CR1 的 M 位定义字长；
- 在 USART_CR2 中编写停止位 STOP 的个数；
- 利用波特率寄存器 USART_BRR 选择希望的波特率；
- 如果需多缓冲器通信，选择 USART_CR3 中的 DMA 使能位(DMAR)。按多缓冲器通信所要求的配置 DMA 寄存器；
- 设置 USART_CR1 的 RE 位。激活接收器，使它开始寻找起始位。

8.6 USART 帧传输协议

串口发送数据时是面向字节的，接收方接收数据时也是面向字节的，实际应用中，收发数据往往都是面向帧(若干字节)的。若一个嵌入式微控制器通过串口往 PC 发送一帧 100 字节数据，由于数据的异步性，字节之间的间隔无法保证，PC 在调用系统 API 来接收数据时，往往不能一次性接收完 100 字节，可能第一次接收 5 字节，第二接收 10 字节，虽然最后都能收到 100 字节。但无法从时间上判断是否是一个完整的数据；如果微控制器发送两帧数据，并且这两帧数据之间的时间间隔非常短，PC 有可能无法区别出两帧数据的边界了，因此我们需要在链路层进行帧格式设计。

典型的串行链路帧协议有点到点传输协议 PPP(Point to Point Protocol)、高级链路控制协议 HDLC(High Data Link Control Protocol)以及面向工业应用的 MODBUS 协议等。

8.6.1 串行链路帧格式设计

为解决异步串行数发送时无法判断多字节流组成的数据帧的头和尾的问题，我们引入两个特殊字符，即帧起始字符 SOF、帧结束字符 EOF，在发送数据时，将多字节数据打包，头部加一个 SOF 字符，尾部加一个 EOF 字符，如图 8-26 所示，这样，接收端检测数据帧的 SOF 字符，一旦检测到 SOF，则认为一个新数据帧的开始，直到接收到 EOF 字符为止。



图 8-26 链路帧格式设计

在串行传输中，有两种传输格式，一种是字符传输，另一种是二进制传输；当采用字符传输时，我们可以定义两个不出现在数据中的特殊字符，如不可见字符制表符、分隔符等，这样可以解决上述帧识别的问题，但在二进制传输中，数据部分可能的取值范围为 0x00～0xFF，因此数据中可能会出现 SOF 或 EOF，这样会出现帧识别错误，如图 8-27 所示。

为解决二进制传输中的问题，我们引入一个转义字符，在发送端的数据中，如果出现了

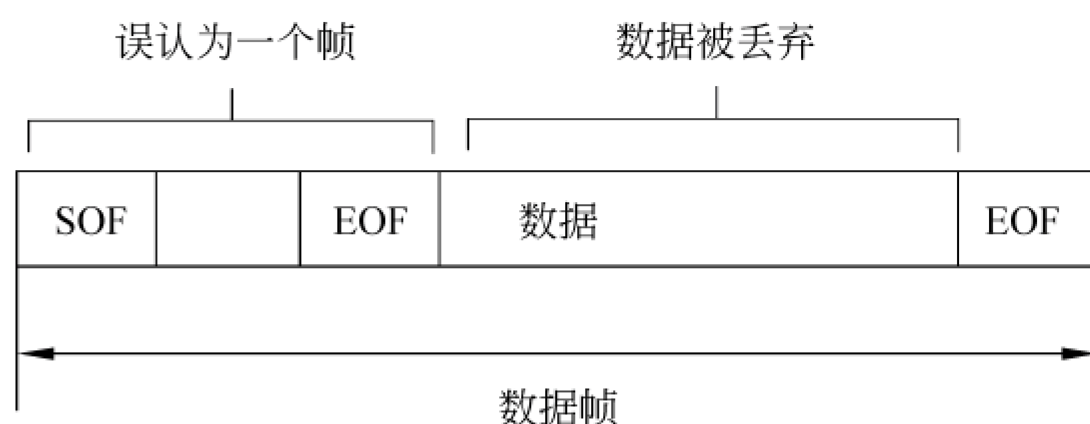


图 8-27　数据中出现 EOF 或 SOF 时帧识别错误

SOF 或 EOF,则插入一个转义字符 ESC 进行变换,若数据中出现了转义字符 ESC,则 ESC 也要进行转义,这样保证发送出去的数据帧中只有一个 SOF 和一个 EOF,接收方接收到数据后进行反转义,恢复数据,转义后的数据格式如图 8-28 所示。

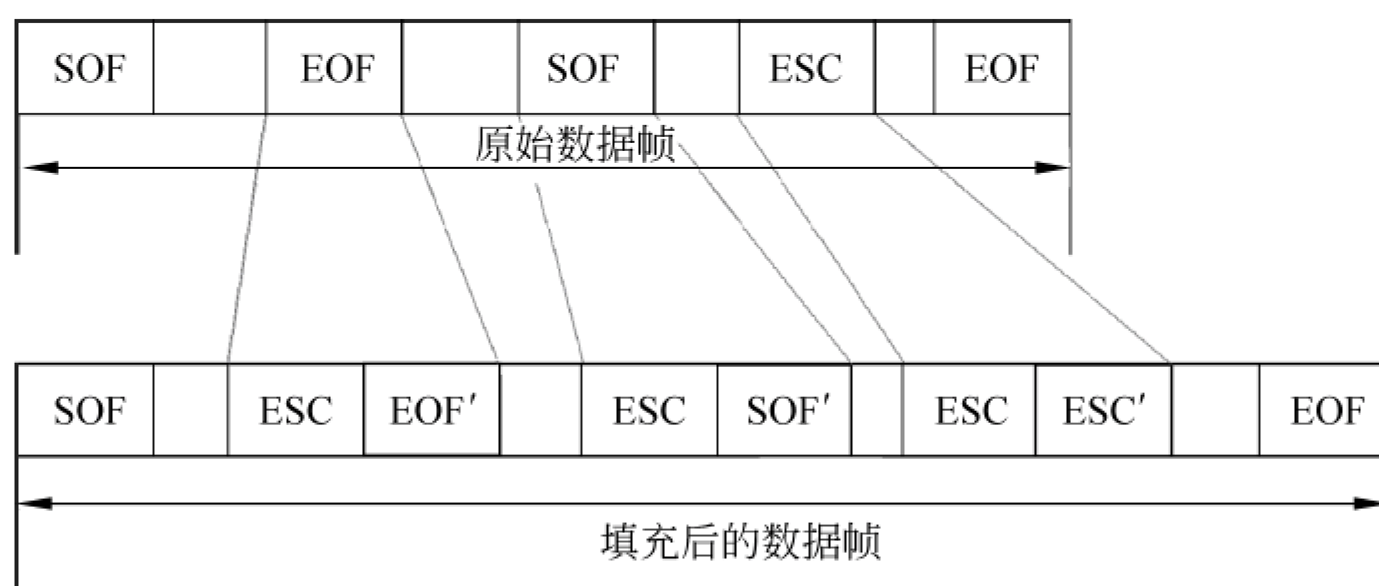


图 8-28　数据转以后的帧格式

1. PPP 协议的字符填充

PPP 协议中,SOF 和 EOF 均取值为 0x7e,ESC 取值为 0x7D,其转义规则如下:

- 如果数据中出现 0x7E,则插入一个 0x7D,将 0x7E 转变为 0x7D 0x7E^0x20 两个字符,即 0x7D 0x5E;
- 如果数据中出现 0x7D,则插入一个 0x7D,将 0x7D 转变为 0x7D 0x7D^0x20 两个字符,即 0x7D 0x5D;
- 如果数据中出现了小于 0x20 的数据,则将该数据转变成 0x7D 原始数据^0x20 两个字符。

^表示异或操作。一个典型例子如图 8-29 所示。

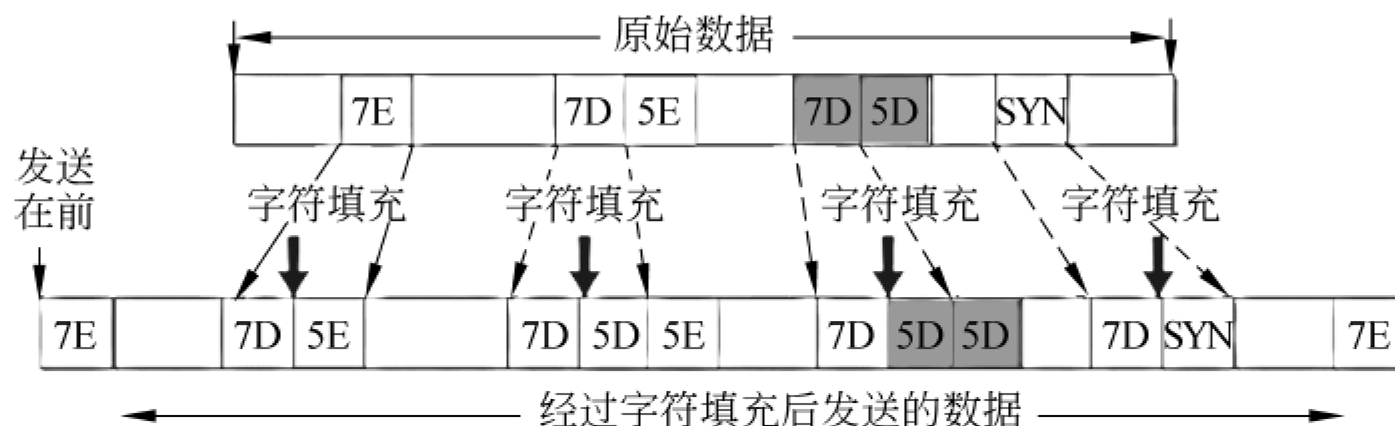


图 8-29　PPP 转义后的数据格式

PPP 数据的编码和解码算法如下：

```
#define PPP_FRAME_FLAG        ( 0x7E )                    /* 标识字符 */
#define PPP_FRAME_ESC         ( 0x7D )                    /* 转义字符 */
#define PPP_FRAME_ENC         ( 0x20 )                    /* 编码字符 */
int ppp_encode(unsigned char * in, int in_len, unsigned char * out, int * out_len)
{
    unsigned char * pi, * po;
    int i, tmp_len;
    pi = in;
    po = out;
    tmp_len = in_len;
    for(i = 0; i < in_len; i++)
    {
        if( * pi == PPP_FRAME_FLAG || * pi == PPP_FRAME_ESC || * pi < 0x20 )
        {
            * po = PPP_FRAME_ESC;
            po++;
            tmp_len++;
            * po = * pi ^ PPP_FRAME_ENC;
        }
        else
            * po = * pi;
        pi++;
        po++;
    }
    * out_len = tmp_len;
    return 0;
}
int ppp_decode(unsigned char * in, int in_len, unsigned char * out, int * out_len)
{
    unsigned char * pi, * po;
    int i, tmp_len;
    pi = in;
    po = out;
    tmp_len = in_len;
    for(i = 0; i < in_len; i++)
    {
        if(* pi == PPP_FRAME_ESC)
        {
            pi++;
            tmp_len--;
            * po = * pi ^ PPP_FRAME_ENC;
```

```
            i++;
        }
        else
            *po = *pi;
        pi++;
        po++;
    }
    *out_len = tmp_len;
    return 0;
}
```

HDLC 协议的转义规则和 PPP 类似，只不过只进行 0x7E 和 0x7D 的转义，不对小于 0x20 的数据转义。除了底层转义外，HDLC 和 PPP 有帧格式定义，如图 8-30 所示。

HDLC

Flag	Address	Control	Data	FCS	Flag
1byte	1Byte	1/2bytes	1500bytes	2/4bytes	1byte

PPP

Flag	Address	Control	Protocol	Data	FCS	Flag
1byte	1Byte	1bytes	1/2bytes	1500bytes	2/4bytes	1byte

图 8-30　HDLC 和 PPP 帧格式定义

Flag 字段即为 SOF 和 EOF 字符，地址字段用来对通信设备进行寻址，Control 字段用于表示控制信息类型，PPP 中的 Protocol 字段用于表示 PPP 报文中封装的 payload(data 字段)的类型，最后的 FCS 字段为校验字节，用于对数据帧进行检错。

2. PPP/HDLC 的串口通信状态机

我们通常利用状态机来实现串行链路帧的发送和接收，图 8-31 是 HDLC 链路帧格式的发送和接收状态机，发送时，初始状态为空闲状态，首先发送 0x7E，进入帧头发送状态，发

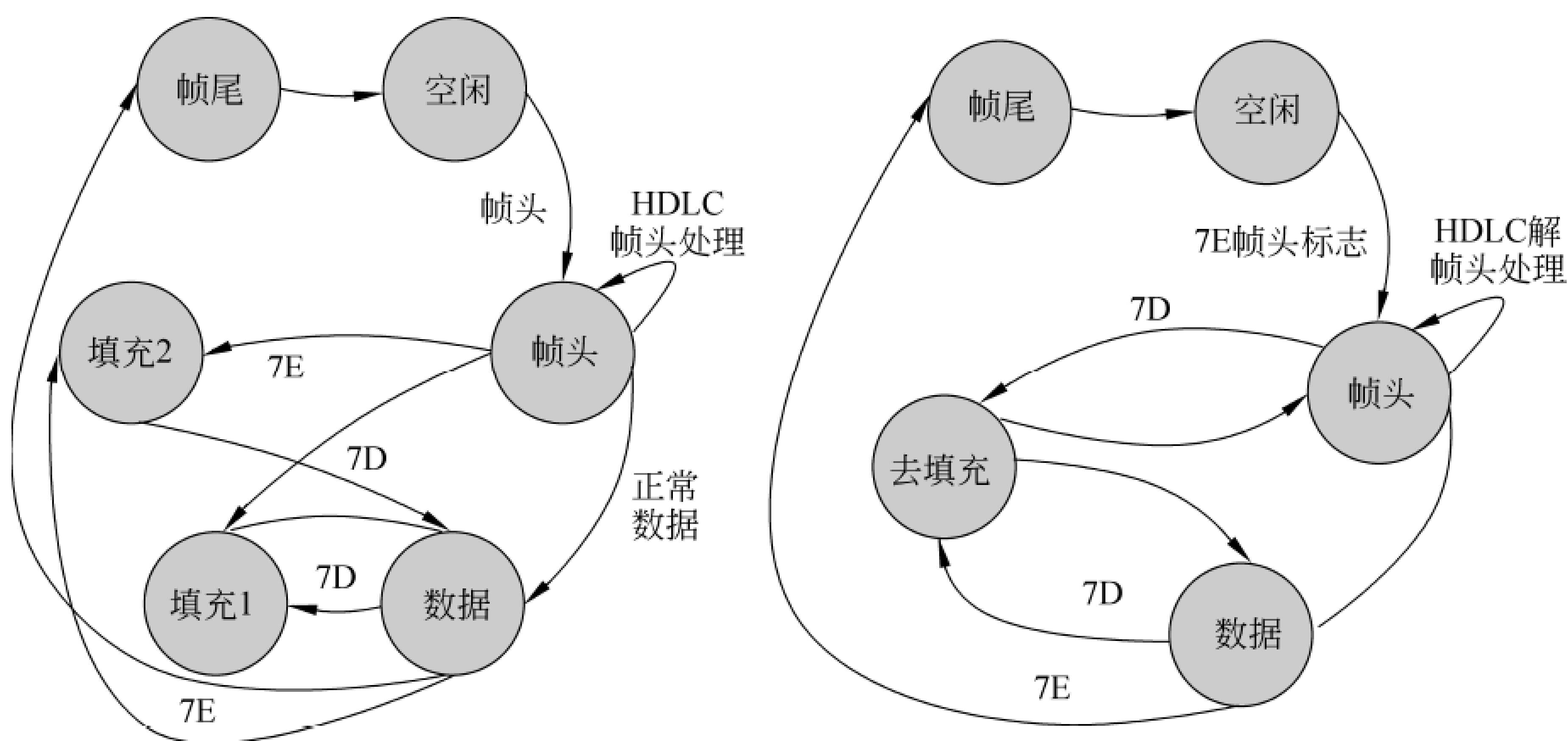


图 8-31　HDLC 串行发送和接收状态机

送完 Address、Control 后，进入数据发送状态（包括了 FCS），在帧头和数据发送过程中，若数据中出现了 0x7D 或 0x7E，则进入转义状态进行转义，发送完所有数据后，发送 0x7E 帧尾结束。

接收时，状态机处于空闲状态，检测到 0x7E，则进入帧头处理状态，紧接着进入数据状态，若数据和帧头中出现了 0x7D 则进入转义状态，接收到 0x7E 后，一个帧接收结束。

8.6.2 MODBUS 帧格式

MODBUS 是一种主从模式的工业现场总线协议，允许一个主机最多连接 247 个从属控制器，支持 RS-485、RS-232、RS-422 和以太网物理层接口，通常用于 PLC，DCS 以及智能仪表的现场总线连接。

MODBUS 的支持 ASCII、RTU 和 TCP 模式，通常使用 ASCII 码模式，每一个数据为一个 ASCII 字符，使用 7b 表示字符，加上 1 个奇偶校验位，串口需配置成 8b 模式，并对串行数据流进行 LRC 校验。另外一种常用的是 RTU 模式，即传输的是二进制数据，采用 CRC 校验，优点是相同传输波特率下，比 ASCII 模式传输数据密度高，速率快，但实现和控制相对较为复杂。

1. ASCII 模式

ASCII 模式下和 HDLC 类似，定义了一个起始字符‘：’和两个结束字符回车 CR、换行 LF 表示数据帧的开始和结尾，所传送数据都是 ASCII 字符，用十六进制表示，即所传输的数据是由 0123456789ABCDEF 等 16 个 ASCII 字符组成的。例如，在发送数据 63 时，则需要发送‘6’和‘3’两个字符。由于数据中不包含‘：’、回车、换行三个字符，因此无需进行转义。

ASCII 模式数据帧格式如图 8-32 所示。

起始字符	Device Address	Function Code	Data	LRC check	结束字符
:	2 字符	2 字符	数个字符	2 字符	2 字符 <CR> <LF>

图 8-32 ASCII 模式数据帧格式

2. RTU 模式

RTU 模式采用的是二进制数据，即每个数据占 8 位，加上一个校验位，串口需要配置成 9b 模式。数据的取值范围为 0x00～0xFF，因此不能使用 ASCII 模式下的起始字符和结束字符。RTU 模式不同于 HDLC 和 PPP，采用了一种类似同步串行的方式，即：RTU 规定每次数据的传输结束，是以未再接到下一个字符间隔时间来判断。其规定为 3.5 字符的通信时间，例如：通信速率为 9600b/s、每个字符含 8b 再加上 1 个起始位及 1 个停止位后，一个字符为 10b。3.5 字符的通信时间为(3.5×10)/9600＝0.00365s，即在 3.65ms 内没有收到数据即认为是数据传输结束。

RTU 模式数据帧格式如图 8-33 所示。

开始间隔	Device Address	Function Code	Data	CRC check	结束间隔
T1-T2-T3-T4	8b	8b	Number of 8b	16b	T1-T2-T3-T4

图 8-33　RTU 模式数据帧格式

8.7　USART 函数库

8.7.1　寄存器定义

USART 寄存器结构 USART_TypeDef 的定义在 stm21L1xx.h 中：

```
typedef struct
{
    __IO uint16_t SR;                                   //状态寄存器
    uint16_t RESERVED1;                                 //保留
    __IO uint16_t DR;                                   //数据寄存器
    uint16_t RESERVED2;                                 //保留
    __IO uint16_t BRR;                                  //波特率因子寄存器
    uint16_t RESERVED3;                                 //保留
    __IO uint16_t CR1;                                  //配置寄存器 1
    uint16_t RESERVED4;                                 //保留
    __IO uint16_t CR2;                                  //配置寄存器 2
    uint16_t RESERVED5;                                 //保留
    __IO uint16_t CR3;                                  //配置寄存器 3
    uint16_t RESERVED6;                                 //保留
    __IO uint16_t GTPR;                                 //保护时间和预分频寄存器
    uint16_t RESERVED7;                                 //保留
} USART_TypeDef;
```

对于 STM32L152，通过如下定义可以确定 3 个 USART 的寄存器地址：

```
#define PERIPH_BASE ((uint32_t)0x40000000)
#define APB1PERIPH_BASE PERIPH_BASE
#define APB2PERIPH_BASE (PERIPH_BASE + 0x10000)
#define AHBPERIPH_BASE (PERIPH_BASE + 0x20000)
#define USART1_BASE (APB2PERIPH_BASE + 0x3800)
#define USART2_BASE (APB1PERIPH_BASE + 0x4400)
#define USART3_BASE (APB1PERIPH_BASE + 0x4800)
```

```
#define USART1 ((USART_TypeDef * ) USART1_BASE)
#define USART2 ((USART_TypeDef * ) USART2_BASE)
#define USART3 ((USART_TypeDef * ) USART3_BASE)
```

ST 提供了 NVIC 标准库函数，头文件位 stm32l1xx_uart.h，程序源代码位于 stm32l1xx_uart.c，USART_InitTypeDef 结构体用于串口的初始化配置，其定义如下：

```
typedef struct
{
    uint32_t USART_BaudRate;
    uint16_t USART_WordLength;
    uint16_t USART_StopBits;
    uint16_t USART_Parity;
    uint16_t USART_Mode;
    uint16_t USART_HardwareFlowControl;
} USART_InitTypeDef;
```

其中，USART_BaudRate 成员设置了 USART 传输的波特率，波特率的取值为 1200～4000000。

USART_WordLength 是一个帧中传输或者接收到的数据位数，其取值为：

- USART_WordLength_8b：8 位数据。
- USART_WordLength_9b：9 位数据。

USART_StopBits 定义了发送的停止位数目，其取值为：

- USART_StopBits_1：在帧结尾传输 1 个停止位。
- USART_StopBits_0.5：在帧结尾传输 0.5 个停止位。
- USART_StopBits_2：在帧结尾传输 2 个停止位。
- USART_StopBits_1.5：在帧结尾传输 1.5 个停止位。

USART_Parity 定义了奇偶模式，其取值为：

- USART_Parity_No：奇偶失能。
- USART_Parity_Even：偶模式。
- USART_Parity_Odd：奇模式。

USART_HardwareFlowControl 指定了硬件流控制模式使能还是失能，其取值为：

- USART_HardwareFlowControl_None：硬件流控制失能。
- USART_HardwareFlowControl_RTS：发送请求 RTS 使能。
- USART_HardwareFlowControl_CTS：清除发送 CTS 使能。
- USART_HardwareFlowControl_RTS_CTS：RTS 和 CTS 使能。

USART_Mode 指定了使能或者失能发送和接收模式，可同时使能发送或接收，其取值为：

- USART_Mode_Tx：发送使能。
- USART_Mode_Rx：接收使能。

USART 同步方式的时钟初始化配置结构体定义如下：

```
typedef struct
{
    uint16_t USART_Clock;
    uint16_t USART_CPOL;
    uint16_t USART_CPHA;
    uint16_t USART_LastBit;
} USART_ClockInitTypeDef;
```

USART_CLOCK 提示了 USART 时钟使能还是失能，其取值为：

- USART_Clock_Enable：时钟高电平活动。
- USART_Clock_Disable：时钟低电平活动。

USART_CPOL：USART_CPOL 指定了下 SLCK 引脚上时钟输出的极性，其取值为：

- USART_CPOL_High：时钟高电平。
- USART_CPOL_Low：时钟低电平。

USART_CPHA 指定了下 SLCK 引脚上时钟输出的相位，和 CPOL 位一起配合来产生不同的时钟/数据的采样关系，其取值为：

- USART_CPHA_1Edge：时钟第一个边沿进行数据捕获。
- USART_CPHA_2Edge：时钟第二个边沿进行数据捕获。

USART_LastBit 来控制是否在同步模式下，在 SCLK 引脚上输出最后发送的那个数据字（MSB）对应的时钟脉冲，其取值为：

- USART_LastBit_Disable：最后一位数据的时钟脉冲不从 SCLK 输出。
- USART_LastBit_Enable：最后一位数据的时钟脉冲从 SCLK 输出。

寄存器初始化结构体可以用来初始化同步串行模式或异步串行模式，每个成员的作用范围如表 8-5 所示。

表 8-5　同步和异步模式下的配置参数

成　　员	异 步 模 式	同 步 模 式
USART_BaudRate	X	X
USART_WordLength	X	X
USART_StopBits	X	X
USART_Parity	X	X
USART_HardwareFlowControl	X	X
USART_Mode	X	X
USART_Clock		X
USART_CPOL		X
USART_CPHA		X
USART_LastBit		X

8.7.2 USART 库函数

ST CMSIS 提供的 USART 主要 API 函数见表 8-6。

表 8-6 USART 主要函数

USART_DeInit	将外设 USARTx 寄存器重设为默认值
USART_Init	根据 USART_InitStruct 中指定的参数初始化外设 USARTx 寄存器
USART_StructInit	把 USART_InitStruct 中的每一个参数按默认值填入
USART_ClockInit	根据 USART_ClockInitStruct 进行时钟相关寄存器初始化
USART_ClockStructInit	把 USART_ClockInitStruct 中的每一个参数按默认值填入
USART_HalfDuplexCmd	使能或者失能 USART 半双工模式
USART_Cmd	使能或者失能 USART 外设
USART_DMACmd	使能或者失能指定 USART 的 DMA 请求
USART_OverSampling8Cmd	使能或失能 8 倍采样时钟
USART_OneBitMethodCmd	使能或失能过采样
USART_SendData	通过外设 USARTx 发送单个数据 USART
USART_ReceiveData	返回 USARTx 最近接收到的数据
USART_ITConfig	使能或者失能指定的 USART 中断
USART_GetFlagStatus	检查指定的 USART 标志位设置与否
USART_ClearFlag	清除 USARTx 的待处理标志位
USART_GetITStatus	检查指定的 USART 中断发生与否
USART_ClearITPendingBit	清除 USARTx 的中断待处理位
USART_SetAddress	设置 USART 节点的地址
USART_WakeUpConfig	选择 USART 的唤醒方式
USART_ReceiverWakeUpCmd	检查 USART 是否处于静默模式

1）USART_DeInit 函数

函数功能：将外设 USARTx 寄存器重设为复位值。

函数原型：void USART_DeInit(USART_TypeDef * USARTx)。

输入参数：USARTx：x 可以是 1,2,3,4,5,来选择 USART 外设。

示例：

```
USART_DeInit(USART1)/                                    /将 USART1 复位
```

2）USART_Init 函数

函数功能：根据 USART_InitStruct 中指定的参数初始化外设 USARTx 寄存器。

函数原型：void USART_Init(USART_TypeDef * USARTx, USART_InitTypeDef * USART_InitStruct)。

输入参数 USARTx：x 可以是 1,2,3,4,5,来选择 USART 外设。

输入参数 USART_InitStruct：指向结构 USART_InitTypeDef 的指针,包含了外设 USART 的配置信息。

示例：

```
USART_InitTypeDef USART_InitStructure;
USART_InitStructure.USART_BaudRate = 9600;
USART_InitStructure.USART_WordLength = USART_WordLength_8b;
USART_InitStructure.USART_StopBits = USART_StopBits_1;
USART_InitStructure.USART_Parity = USART_Parity_Odd;
USART_InitStructure.USART_HardwareFlowControl =
                                        USART_HardwareFlowControl_RTS_CTS;
USART_InitStructure.USART_Mode = USART_Mode_Tx | USART_Mode_Rx;
USART_Init(USART1, &USART_InitStructure);
```

3）USART_StructInit 函数

函数功能：把 USART_InitStruct 中的每一个参数按默认值填入。

函数原型：void USART_StructInit(USART_InitTypeDef * USART_InitStruct)。

输入参数：USART_InitStruct：指向结构 USART_InitTypeDef 的指针,待初始化。

USART_InitStruct 默认值为：

```
USART_BaudRate:                 9600
USART_WordLength:               USART_WordLength_8b
USART_StopBits:                 USART_StopBits_1
USART_Parity:                   USART_Parity_No
USART_HardwareFlowControl:      USART_HardwareFlowControl_None
USART_Mode:                     USART_Mode_Rx | USART_Mode_Tx
```

示例：

```
USART_InitTypeDef USART_InitStructure;
USART_StructInit(&USART_InitStructure);
```

4）USART_ClockInit 函数

函数功能：根据 USART_ClockInitStruct 参数初始化 USARTx 的时钟配置。

函数原型：void USART_ClockInit (USART_TypeDef * USARTx, USART_ClockInitTypeDef * USART_ClockInitStruct)。

输入参数 USARTx,用于指定 USART,x 取值范围为 1,2,3。

输入参数 USART_ClockInitStruct,指向 USART_ClockInitTypeDef 的指针,包含了

USART 同步模式下的时钟配置信息。

示例：

```
USART_ClockInitTypeDef* USART_ClockInitStruct;
USART_ClockInitStructure.USART_Clock = USART_Clock_Disable;
USART_ClockInitStructure.USART_CPOL = USART_CPOL_High;
USART_ClockInitStructure.USART_CPHA = USART_CPHA_1Edge;
USART_ClockInitStructure.USART_LastBit = USART_LastBit_Enable;
USART_ClockInit(&USART_ClockInitStructure);
```

5）USART_ClockStructInit 函数

函数功能：把 USART_ClockInitStruct 中的每一个参数按默认值填入。

函数原型：void USART_ClockStructInit(USART_ClockInitTypeDef * SART_ClockInitStruct)。

输入参数 USART_ClockInitStruct：指向 USART_ClockInitTypeDef 的指针，待初始化。

USART_InitStruct 默认值为：

```
USART_Clock       USART_Clock_Disable
USART_CPOL        USART_CPOL_Low
USART_CPHA        USART_CPHA_1Edge
USART_LastBit     USART_LastBit_Disable
```

示例：

```
USART_ClockInitTypeDef* USART_ClockInitStruct;
USART_ClockStructInit(&USART_ClockInitStructure);
```

6）USART_OverSampling8Cmd 函数

函数功能：启用或禁用 USART 的 8 倍采样时钟模式。

函数原型：void USART_OverSampling8Cmd(USART_TypeDef * USARTx, FunctionalState NewState)。

输入参数 USARTx：用于指定待配置的串口，x 的取值范围为 1,2,3,4,5。

输入参数 NewState：用于使能或禁用 8 倍采样时钟，取值为 ENABLE 或 DISABLE。

该函数必须在 USART_Init 之前调用。示例如下：

```
USART_OverSampling8Cmd(USART1,ENABLE);
```

7）USART_OneBitMethodCmd 函数

函数功能：用于使能或禁用过采样功能。

函数原型：void USART_OneBitMethodCmd(USART_TypeDef * USARTx, FunctionalState NewState)。

输入参数 USARTx：用于指定待配置的串口，x 的取值范围为 1,2,3,4,5。

输入参数 NewState：用于使能或禁用过采样，取值为 ENABLE 或 DISABLE。

示例：

```
USART_OneBitMethodCmd(USART1,ENABLE);
```

8）USART_Cmd 函数

函数功能：使能或者失能 USART 外设。

函数原型：void USART_Cmd(USART_TypeDef * USARTx，FunctionalState NewState)。

输入参数 USARTx：x 可以是 1，2 或者 3，来选择 USART 外设。

输入参数 NewState：外设 USARTx 的新状态，参数取值为：ENABLE 或者 DISABLE。

示例：

```
USART_Cmd(USART1, ENABLE);
```

9）USART_DMACmd 函数

函数功能：使能或者失能指定 USART 的 DMA 请求。

函数原型：USART_DMACmd(USART_TypeDef * USARTx，uint16_t USART_DMAReq，FunctionalState NewState)。

输入参数 USARTx：用于指定待配置的串口，x 的取值范围为 1，2，3，4，5。

输入参数 USART_DMAreq：指定 DMA 请求，取值为：

- USART_DMAReq_Tx 发送 DMA 请求。
- USART_DMAReq_Rx 接收 DMA 请求。

输入参数 NewState：USARTx DMA 请求源的新状态，取值为 ENABLE 或者 DISABLE。

示例：

```
USART_DMACmd(USART2, USART_DMAReq_Rx | USART_DMAReq_Tx, ENABLE);
```

10）USART_SendData 函数

函数功能：发送一个字节的数据。

函数原型：void USART_SendData(USART_TypeDef * USARTx，uint16_t Data)。

输入参数 USARTx：用于指定待配置的串口，x 的取值范围为 1，2，3，4，5。

输入参数 Data：待发送的数据。

示例：

```
USART_SendData(USART3, 0x26);
```

11）USART_ReceiveData 函数

函数功能：返回 USARTx 最近接收到的数据。

函数原型：uint16_t USART_ReceiveData(USART_TypeDef * USARTx)。

输入参数 USARTx：用于指定待配置的串口，x 的取值范围为 1，2，3，4，5。

返回值：接收到的数据。

示例：

```
uint16_t RxData = USART_ReceiveData(USART2);
```

12）USART_ITConfig 函数

函数功能：使能或者失能指定的 USART 中断。

函数原型：void USART_ITConfig(USART_TypeDef * USARTx，uint16_t USART_IT，FunctionalState NewState)。

输入参数 USARTx：用于指定待配置的串口，x 的取值范围为 1，2，3，4，5。

输入参数 USART_IT：待使能或者失能的 USART 中断源，其取值为：

- USART_IT_PE　　奇偶错误中断
- USART_IT_TXE　　发送中断
- USART_IT_TC　　传输完成中断
- USART_IT_RXNE　　接收中断
- USART_IT_IDLE　　空闲总线中断
- USART_IT_LBD　　LIN 中断检测中断
- USART_IT_CTS　　CTS 中断
- USART_IT_ERR　　错误中断

输入参数 NewState：外设 USARTx 的新状态，参数取值为 ENABLE 或者 DISABLE。

示例：

```
USART_ITConfig(USART1, USART_IT_Transmit, ENABLE);
```

13）USART_GetFlagStatus 函数

函数功能：检查指定的 USART 标志位设置与否。

函数原型：FlagStatus USART_GetFlagStatus(USART_TypeDef * USARTx，uint16_t USART_FLAG)。

输入参数 USARTx：用于指定待配置的串口，x 的取值范围为 1，2，3，4，5。

输入参数 USART_FLAG：待检查的 USART 标志位，取值为：

- USART_FLAG_CTS　　CTS 标志位
- USART_FLAG_LBD　　LIN 中断检测标志位
- USART_FLAG_TXE　　发送数据寄存器空标志位
- USART_FLAG_TC　　发送完成标志位
- USART_FLAG_RXNE　　接收数据寄存器非空标志位
- USART_FLAG_IDLE　　空闲总线标志位
- USART_FLAG_ORE　　溢出错误标志位
- USART_FLAG_NE　　噪声错误标志位
- USART_FLAG_FE　　帧错误标志位
- USART_FLAG_PE　　奇偶错误标志位

返回值：待检查的标志位的状态,SET 或 RESET。

示例：

```
FlagStatus Status = USART_GetFlagStatus(USART1, USART_FLAG_TXE);
```

14）USART_ ClearFlag 函数

函数功能：清除 USARTx 的待处理标志位。

函数原型：void USART_ClearFlag(USART_TypeDef * USARTx，uint16_t USART_FLAG)。

输入参数 USARTx：用于指定待配置的串口,x 的取值范围为 1,2,3,4,5。

输入参数 USART_FLAG：待清除的 USART 标志位,取值和 USART_GetFlagStatus 的 USART_FLAG 相同。

示例：

```
USART_ClearFlag(USART1,USART_FLAG_OR);
```

15）USART_ GetITStatus 函数

函数功能：检查指定的 USART 中断发生与否。

函数原型：ITStatus USART_GetITStatus(USART_TypeDef * USARTx，uint16_t USART_IT)。

输入参数 USARTx：用于指定待配置的串口,x 的取值范围为 1,2,3,4,5。

输入参数 USART_IT：待检查的 USART 中断源,取值为：

- USART_IT_PE　　奇偶错误中断
- USART_IT_TXE　　发送中断
- USART_IT_TC　　发送完成中断
- USART_IT_RXNE　　接收中断
- USART_IT_IDLE　　空闲总线中断
- USART_IT_LBD　　LIN 中断探测中断
- USART_IT_CTS　　CTS 中断
- USART_IT_ORE　　溢出错误中断
- USART_IT_NE　　噪音错误中断
- USART_IT_FE　　帧错误中断

返回值：USART_IT 的新状态,SET 或 RESET。

示例：

```
ITStatus ErrorITStatus = USART_GetITStatus(USART1, USART_IT_ORE);
```

16）USART_ ClearITPendingBit 函数

函数功能：清除 USARTx 的中断待处理位。

函数原型：void USART_ClearITPendingBit(USART_TypeDef * USARTx，uint16_t

USART_IT)。

输入参数 USARTx：用于指定待配置的串口，x 的取值范围为 1，2，3，4，5。

输入参数 USART_IT：待检查的 USART 中断源，取值和 USART_ GetITStatus 函数 USART_IT 参数相同。

示例：

```
USART_ClearITPendingBit(USART1,USART_IT_OverrunError);
```

17）USART_SetAddress 函数

函数功能：设置 USART 设备的地址，用于多主机通信。

函数原型：void USART_SetAddress(USART_TypeDef * USARTx，uint8_t USART_Address)。

输入参数 USARTx：用于指定待配置的串口，x 的取值范围为 1，2，3，4，5。

输入参数 USART_Address：USART 设备的地址。

示例：

```
USART_SetAddress(USART2, 0x5);
```

18）USART_WakeUpConfig 函数

函数功能：选择 USART 的唤醒方式，用于多主机通信。

函数原型：void USART_WakeUpConfig(USART_TypeDef * USARTx，uint16_t USART_WakeUp)。

输入参数 USARTx：用于指定待配置的串口，x 的取值范围为 1，2，3，4，5。

输入参数 USART_WakeUp：USART 的唤醒方式，取值为：

- USART_WakeUp_IdleLine　　空闲总线唤醒
- USART_WakeUp_AddressMark　　地址标记唤醒

示例：

```
USART_WakeUpConfig(USART1, USART_WakeUpIdleLine);
```

19）USART_ReceiverWakeUpCmd 函数

函数功能：检查 USART 是否处于静默模式，用于多主机通信。

函数原型：void USART_ReceiverWakeUpCmd(USART_TypeDef * USARTx，FunctionalState Newstate)。

输入参数 USARTx：用于指定待配置的串口，x 的取值范围为 1，2，3，4，5。

输入参数 NewState：USART 静默模式的新状态，参数取值为 ENABLE 或者 DISABLE。

示例：

```
USART_ReceiverWakeUpCmd(USART3, DISABLE);
```

8.8　USART 案例

8.8.1　串口寄存器操作案例

1）串口初始化函数 USART_Init 的实现

```
void USART_Init(USART_TypeDef * USARTx, USART_InitTypeDef * USART_InitStruct)
{
  uint32_t tmpreg = 0x00, apbclock = 0x00;
  uint32_t integerdivider = 0x00;
  uint32_t fractionaldivider = 0x00;
  RCC_ClocksTypeDef RCC_ClocksStatus;
  tmpreg = USARTx->CR2;
  tmpreg &= (uint32_t)~((uint32_t)USART_CR2_STOP); //清除 STOP 域
  //根据参数 USART_StopBits 设置 STOP 域 value
  tmpreg |= (uint32_t)USART_InitStruct->USART_StopBits;
  //将配置参数写入控制寄存器 2
  USARTx->CR2 = (uint16_t)tmpreg;
  tmpreg = USARTx->CR1;
  //清除 M、PCE、PS、TE 和 RE 域
  tmpreg &= (uint32_t)~((uint32_t)CR1_CLEAR_MASK);
  //根据输入参数设置 M、PCE、PS、TE 和 RE 域
  tmpreg   |= (uint32_t)USART_InitStruct->USART_WordLength
           | USART_InitStruct->USART_Parity |USART_InitStruct->USART_Mode;
  //将配置参数写入控制寄存器 1
  USARTx->CR1 = (uint16_t)tmpreg;
  //配置控制寄存器 3
  tmpreg = USARTx->CR3;
  //清除 CTSE 和 RTSE
  tmpreg &= (uint32_t)~((uint32_t)CR3_CLEAR_MASK);
  //根据输入参数 USART_HardwareFlowControl 配置 CTSE 和 RTSE 域
  tmpreg |= USART_InitStruct->USART_HardwareFlowControl;
  USARTx->CR3 = (uint16_t)tmpreg;
  //波特率寄存器配置,首先获取总线时钟
  RCC_GetClocksFreq(&RCC_ClocksStatus);
  //USART1 连接在 APB2,获取 APB2 总线时钟
  if (USARTx == USART1)
      apbclock = RCC_ClocksStatus.PCLK2_Frequency;
  else //其余连接在 APB1,获取 APB1 总线时钟
```

```
        apbclock = RCC_ClocksStatus.PCLK1_Frequency;
    if ((USARTx->CR1 & USART_CR1_OVER8) != 0)
    //8倍波特率采样时钟配置下计算整数部分
     integerdivider= ((25* apbclock) / (2*  (USART_InitStruct->USART_BaudRate)));
    else //16倍波特率采样时钟配置下计算整数部分
    integerdivider= ((25* apbclock) / (4 *  (USART_InitStruct->USART_BaudRate)));
    tmpreg = (integerdivider / 100) << 4;
    //小数部分计算
    fractionaldivider = integerdivider - (100 *  (tmpreg >> 4));
    //8倍波特率采样时钟
    if ((USARTx->CR1 & USART_CR1_OVER8) != 0)
        tmpreg |= ((((fractionaldivider *  8) + 50) / 100)) & ((uint8_t)0x07);
    else //16倍波特率采样时钟
      tmpreg |= ((((fractionaldivider *  16) + 50) / 100)) & ((uint8_t)0x0F);
     //写入波特率寄存器
    USARTx->BRR = (uint16_t)tmpreg;
}
```

2）USART_SendData 函数的实现

```
void USART_SendData(USART_TypeDef*  USARTx, uint16_t Data)
{
   //数据最长为 9b,因此与 0x1FF 进行位与操作
   USARTx->DR = (Data & (uint16_t)0x01FF);
}
```

8.8.2 串口配置基本流程

1）配置 I/O

```
//设置 Tx 引脚为推拉输出模式
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9 | GPIO_Pin_2;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_40MHz;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
GPIO_InitStructure.structure.GPIO_OType= _GPIO_OType_PP;
GPIO_Init(GPIOA, &GPIO_InitStructure);
//设置 Rx 引脚
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10 | GPIO_Pin_3;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
GPIO_InitStructure.GPIO_Pupd = GPIO_Pupd_Nopull;
GPIO_Init(GPIOA, &GPIO_InitStructure);
RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1, ENABLE);
```

2）配置 UART

```
void USART3_Configuration(void)
{
    USART_InitTypeDef USART_InitStructure;
    USART_InitStructure.USART_BaudRate = 115200;                          //设置波特率
    USART_InitStructure.USART_WordLength = USART_WordLength_8b;           //数据长度 8 位
    USART_InitStructure.USART_StopBits = USART_StopBits_1;                //一个停止位
    USART_InitStructure.USART_Parity = USART_Parity_No;                   //无奇偶校验
    USART_InitStructure.USART_HardwareFlowControl
                    = USART_HardwareFlowControl_None;                     //无非硬件流控制
    USART_InitStructure.USART_Mode= USART_Mode_Tx|USART_Mode_Rx;
    //允许接收和发送
    USART_Init(USART3,&USART_InitStructure);
    //配置接收中断
    USART_ITConfig(USART3, USART_IT_RXNE, ENABLE);
    //使能串口
    USART_Cmd(USART3, ENABLE);
}
```

3）配置 NVIC

```
void NVIC_Configuration(void)
//使能串口中断,同时要设置中断的优先级
NVIC_InitStructure.NVIC_IRQChannel= USART1_IRQn;
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority= 0;
NVIC_InitStructure.NVIC_IRQChannelSubPriority= 0;
NVIC_InitStructure.NVIC_IRQChannelCmd= ENABLE;
//使能串口中断
NVIC_Init(&NVIC_InitStructure)
```

4）中断函数 USART1_IRQHandler

```
Void USART3_IRQHandler(void)
{
    Unsigned char k= 0,buf1= 0;
    if(USART_GetITStatus(USART3,USART_IT_RXNE))
    {   //判断是否为接收数据中断
        buf1= USART_ReceiveData(USART3);
        USART_ClearITPendingBit(USART3,USART_FLAG_TC);
    }
}
```

8.8.3　PC 串口通信案例

1）查询方式发送数据

```
void USART_SendString(uint8_t * in)
```

```
{
    for(uint8_t i=0;in[i]!='\0';i++)
    {
        USART_Send(USART2,in[i]);
        while (USART_GetFlagStatus(USART2, USART_FLAG_TC) ==RESET);
    }
}
```

2）中断方式发送数据

```
uint8_t CmdBuffer[3]={'a','b','c'};
uint8_t TxIndex=0;
USART_ITConfig(USARTx, USART_IT_TXE, ENABLE);
while(1);
```

中断服务程序：

```
void USARTx_IRQHandler(void)
{
    if (USART_GetITStatus(USARTx, USART_IT_TXE) ==SET)
    {
        USART_SendData(USARTx, CmdBuffer[TxIndex++]);
        if (TxIndex ==0x03)
            USART_ITConfig(USARTx, USART_IT_TXE, DISABLE);
}
```

3）串口 printf 输出

```
#include<stdio.h>
int main(void)
{
  //配置串口
  USART_InitStructure.USART_BaudRate =115200;
  USART_InitStructure.USART_WordLength =USART_WordLength_8b;
  USART_InitStructure.USART_StopBits =USART_StopBits_1;
  USART_InitStructure.USART_Parity =USART_Parity_No;
  USART_InitStructure.USART_HardwareFlowControl=
                                                    USART_HardwareFlowControl_None;
  USART_InitStructure.USART_Mode =USART_Mode_Rx | USART_Mode_Tx;
  //TX RX GPIO配置
  GPIO_InitTypeDef GPIO_InitStructure;
  RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOD, ENABLE);           //使能 UART时钟
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_USART2, ENABLE);        //配置 USART_Tx引脚
  GPIO_PinAFConfig(GPIOD, GPIO_Pin Source 5, GPIO_AF_USART2);   //配置 USART_Rx引脚
  GPIO_PinAFConfig(GPIOD, GPIO_Pin Source 6, GPIO_AF_USART2);
```

```
    //配置 USART Tx I/O 复用推挽
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_40MHz;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
    GPIO_Init(GPIOD, &GPIO_InitStructure);
    //配置 USART Rx I/O 复用推挽
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6;
    GPIO_Init(GPIOD, &GPIO_InitStructure);
    USART_Init(USART2, USART_InitStruct);                        //初始化串口
    USART_Cmd(USART2, ENABLE);                                   //使能串口
    //调用 printf 输出
    printf("\n\rUSART Printf Example\n\r");
    //判断 TC,等待 printf 传输完成
    while (USART_GetFlagStatus(USART2, USART_FLAG_TC) == RESET);
    while (1);
}
//重定向 printf 的输出
int fputc(int ch, FILE * f)
{
    USART_SendData(USART2, (uint8_t) ch);
    //等待数据寄存器为空,可继续写入下一个字符
    while (USART_GetFlagStatus(EVAL_COM1, USART_FLAG_TXE) == RESET);
    return ch;
}
```

8.8.4　状态机多字节数据帧发送和接收案例

数据通信规范采用 HDLC 规范进行数据发送，数据格式为：

7e	type	data	crc1	crc2	7e

发送数据定义：

```
typedef struct _TxMsg {
    uint8_t type;
    uint8_t data1;
    uint8_t data2;
} Tx_Msg ;
typedef Tx_Msg*  Tx_MsgPtr;
```

接收数据定义：

```
typedef struct _MsgRcvEntry {
    uint8_t Length;
    Tx_Msg Msg;
    uint16_t CRC;
} Rx_Msg;
```

发送和接收数据缓冲区：

```
Tx_Msg gSendBuf={65,1,2};
Rx_Msg gRcvBuf;
```

转义字符及数据中的 type 类型定义：

```
enum {
    HDLC_MTU            = (sizeof(Tx_Msg)),
    HDLC_FLAG_BYTE      = 0x7e,
    HDLC_CTLESC_BYTE    = 0x7d,
    PROTO_ACK           = 64,
    PROTO_PACKET_ACK    = 65,
    PROTO_PACKET_NOACK = 66,
    PROTO_UNKNOWN       = 255
};
```

发送状态机状态定义：

```
enum {
    TXSTATE_IDLE,
    TXSTATE_TYPE,
    TXSTATE_DATA,
    TXSTATE_ESC,
    TXSTATE_FCS1,
    TXSTATE_FCS2,
    TXSTATE_ENDFLAG,
    TXSTATE_FINISH,
    TXSTATE_ERROR
};
```

接收状态机状态定义：

```
enum {
    RXSTATE_NOSYNC,
    RXSTATE_TYPE,
    RXSTATE_DATA,
    RXSTATE_ESC
  };
```

状态机和变量初始化：

```
gRcvBuf.Length = 0;
uint8_t gTxState = TXSTATE_IDLE;
uint8_t gTxByteCnt = 0;
uint8_t gTxLength = 0;
uint16_t gTxRunningCRC = 0;
uint8_t* gpsend = (uint8_t* ) (&gSendBuf);
uint8_t gRxState = RXSTATE_NOSYNC;
uint8_t gRxHeadIndex = 0;
uint8_t gRxTailIndex = 0;
uint8_t gRxByteCnt = 0;
uint16_t gRxRunningCRC = 0;
uint8_t* gpRxBuf = (uint8_t* ) (& gRcvBuf.Tx_Msg);
```

主函数：

```
main() {
  //配置串口
  usart2config();
  //发送 0x7E,使能 TC 中断
  gTxLength = sizeof(gSendBuf);
  gTxState = TXSTATE_TYPE;
  USART_Send(USART2,HDLC_FLAG_BYTE);
  USART_IT_Config(USART2, USART_IT_TC,ENABLE);
  USART_IT_Config(USART2, USART_IT_RXNE,ENABLE);
  while(1);
}
```

中断处理函数：

```
void USARTx_IRQHandler(void) {
  uint8_t nextByte= 0;
  if (USART_GetITStatus(USART2, USART_IT_TC) == = SET) {
    switch (gTxState) {
    case TXSTATE_TYPE:
      gTxState = TXSTATE_DATA;
      nextByte = * gpsend+ + ;
      gTxRunningCRC = crcByte(gTxRunningCRC, nextByte);
      USART_Send(USART2, nextByte);
      gTxByteCnt+ +
      break;
    case TXSTATE_DATA:
      nextByte = * gpsend+ + ;
      gTxRunningCRC = crcByte(gTxRunningCRC,nextByte);
```

```
            gTxByteCnt++;
            if (gTxByteCnt >= gTxLength)
              gTxState = TXSTATE_FCS1;
            TxArbitraryByte(nextByte);
            break;
          case TXSTATE_ESC:
            TxResult = USART_Send(USART2, (gTxEscByte ^ 0x20));
            gTxState = gPrevTxState;
            break;
          case TXSTATE_FCS1:
            nextByte = (uint8_t)(gTxRunningCRC & 0xff);                    // LSB
            gTxState = TXSTATE_FCS2;
            TxArbitraryByte(nextByte);
            break;
          case TXSTATE_FCS2:                                               // MSB
            nextByte = (uint8_t)((gTxRunningCRC >> 8) & 0xff);
            gTxState = TXSTATE_ENDFLAG;
            TxArbitraryByte(nextByte);
            break;
          case TXSTATE_ENDFLAG:
            gTxState = TXSTATE_FINISH;
            TxResult = USART_Send(USART2,HDLC_FLAG_BYTE);
            break;
          case TXSTATE_FINISH:
          case TXSTATE_ERROR:
          default:
            break;
        }
      }
      if (USART_GetITStatus(USART2, USART_IT_RXNE) == SET){
        uint8_t data =   USART_Receive(USART2);
        switch (gRxState) {
          case RXSTATE_NOSYNC:
          if ((data == HDLC_FLAG_BYTE) && (gRcvBuf.Length == 0)) {
                gRxByteCnt = gRxRunningCRC = 0;
                gRxState = RXSTATE_TPYE;
            }
            break;
          case RXSTATE_TPYE:
            * gpRxBuf++ = data;
            gRxRunningCRC = crcByte(gRxRunningCRC,data);
            gRxState = RXSTATE_DATA;
            gRxByteCnt++;
```

```
        break;
      case RXSTATE_DATA:
        if (gRxByteCnt > HDLC_MTU) {
            gRxByteCnt = gRxRunningCRC = 0;
            gRcvBuf.Length = 0;
            gRxState = RXSTATE_NOSYNC;
         }
         else if (data == HDLC_CTLESC_BYTE)
            gRxState = RXSTATE_ESC;
         else if (data == HDLC_FLAG_BYTE) {                              //收到结束字符
            if (gRxByteCnt >= 2) {
              uint16_t usRcvdCRC = (gpRxBuf[(gRxByteCnt-1)] & 0xff);
              usRcvdCRC = (usRcvdCRC << 8) | (gpRxBuf[(gRxByteCnt-2)] & 0xff);
                if (usRcvdCRC == gRxRunningCRC) {                        //校验
                gRcvBuf.Length = gRxByteCnt - 2;
                //PacketRcvd();判断接收数据进行下一步处理
                }
                 else
                   gRcvBuf.Length = 0;
             }
             else {                                                      //不够数
                  gRcvBuf.Length = 0;
                  gRxState = RXSTATE_NOSYNC;
             }
             gRxByteCnt = gRxRunningCRC = 0;
         }
        else {
            * gpRxBuf++ = data;
            if (gRxByteCnt >= 2)
                gRxRunningCRC = crcByte(gRxRunningCRC,gpRxBuf[(gRxByteCnt-2)]);
            gRxByteCnt++;
      }
      break;
   }
   case RXSTATE_ESC:
     if (data == HDLC_FLAG_BYTE) {
        gRxByteCnt = gRxRunningCRC = 0;
        gMsgRcvTbl[gRxHeadIndex].Length = 0;
        gMsgRcvTbl[gRxHeadIndex].Token = 0;
        gRxState = RXSTATE_NOSYNC;
   }
   else {
     data = data ^ 0x20;
```

```
            gpRxBuf[gRxByteCnt] = data;
            if (gRxByteCnt >= 2) {
            gRxRunningCRC = crcByte(gRxRunningCRC,gpRxBuf[(gRxByteCnt-2)]);
        }
        gRxByteCnt++;
        gRxState = RXSTATE_INFO;
        }
        break;
    default:
            gRxState = RXSTATE_NOSYNC;
            break;
    }
        return SUCCESS;
}
//转义发送函数
uint8_t TxArbitraryByte(uint8_t Byte) {
        if ((Byte == HDLC_FLAG_BYTE) || (Byte == HDLC_CTLESC_BYTE)) {
            gPrevTxState = gTxState;
            gTxState = TXSTATE_ESC;
            gTxEscByte = Byte;
            Byte = HDLC_CTLESC_BYTE;
        }
            USART_Send(USART2,Byte);
        return 1;
    }
//crc函数
uint16_t crcByte(uint16_t crc,uint8_t data)
{
        //此处省略CRC计算函数
}
```

Chapter 9

第9章　IIC总线

【导读】 IIC总线常用于微控制器和数字外设芯片之间的连接,是一种典型的同步总线,本章首先介绍了IIC总线的时序,然后对STM32L152的IIC总线控制器的内部结构,寄存器以及不同模式的发送和接收配置流程进行了介绍,最后介绍了CMSIS提供的典型寄存器操作库函数。针对IIC总线时序传输,本章还对典型外设Flash、温度传感器的操作时序进行了案例说明。

9.1　IIC总线概述

IIC(Inter Integrated-Circuit,也记为I^2C或I2C)总线是由PHILIPS公司1982年提出的一种用于连接微控制器和低速外设的短距离串行总线标准,通信速率从最初的100kHz到2016年I2C第六版高达5MHz。I2C是两线制总线,由一根时钟线SCL和一个数据线SDA组成。

I2C是支持多主设备和多从设备的单工总线,总线硬件连接简单,将不同I2C设备的SCL和SDA直接相连即可,在多设备的使用中,I2C总线采用地址寻址方法识别索要操作的设备,避免了片选寻址的弊端,从而使硬件系统扩展更为灵活。I2C协议简单,适用于低成本芯片作为接口,典型的I2C接口外设有:串行存储器、低速ADC和DAC、RTC以及LCD控制等。在I2C的基础上,衍生出来的子标准有系统管理总线SMBus、电源管理总线PMBus、智能平台管理接口IPMI、显示数据通道DDC和高级电信计算架构ATCA等。

I2C是单工通信总线,由主设备和从设备构成,任何能够进行发送和接收的设备都可以成为主设备,主设备能够产生时钟,发起和结束一次数据传输。一般应用中,一个主设备控制多个从设备,I2C支持多个主设备工作,在总线上可以有多个主设备发起通信,若产生冲突,通过总线仲裁进行解决,即在任何时间点只能有一个主设备处于通信状态。

I2C总线的主要优势如下:

- 两线制总线,占用I/O数量少;
- 总线上的所有设备通过软件寻址,每个设备具有唯一的地址;
- 设备连接为主/从关系,主机可以是主发送器或主接收器;
- 通过冲突检测和仲裁机制支持多主机通信;
- 总线传输带有ACK和NACK应答,能保证数据传输的可靠性;

- 支持多种速率：标准模式(Standard Mode)100kb/s、快速模式(Fast Mode)400kb/s、增强快速模式(Fast Mode Plus)1Mb/s 和高速模式(High Speed Mode)3.4Mb/s；极速模式(Ultra-Fast Mode)，单向数据传输速率可达 5Mb/s。

I2C 只有数据和时钟两条线，在处理地址和应答时存在一定的开销，效率不如设备直接相连的 SPI 总线。I2C 的数据(SDA)和时钟(SCL)信号都是双向的，通过上拉电阻接到电源(如图 9-1 所示)。两根线都为高电平时，总线处于空闲状态(IDLE)。I2C 接口通过线与功能实现多设备的总线连接，总线的每个信号接口都包括一个开漏输出和输入缓冲器。开漏电路不能输出高电平，因此必须通过外接上拉电阻输出高电平。如果总线上有任何一个设备接口输出低电平，则整个总线的状态表现为低电平，体现出逻辑与的特点，即 I2C 的线与功能。线与功能的好处在于可以实现总线的仲裁控制。总线的控制权会交给最后一个输出低电平的设备，其他设备(输出为高)通过检测总线上的电平状态(表现为低)，对比与自己输出状态不一致，则自动退出对总线的控制请求。

开漏电路不适合长距离通信，信号线越长，信号的反射和振荡越强，从而影响总线的信号完整性，总线速度越快，对于线上干扰的要求越高，I2C 只适合电路板级的短距离通信。

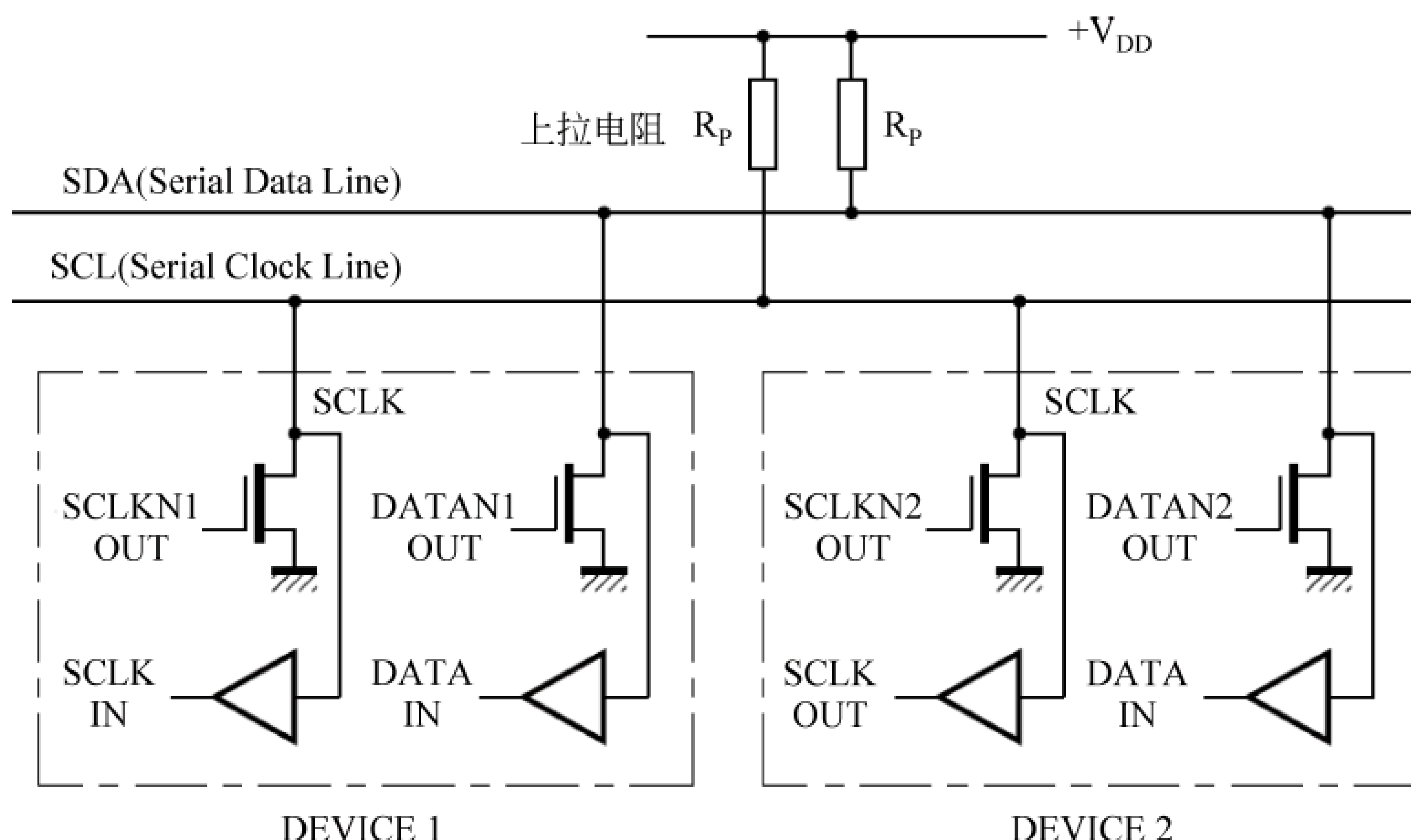


图 9-1　I2C 连接示意图

9.2　I2C 总线的基本操作

1. 数据有效性

I2C 协议一般采用 3V 或 5V 作为高电平 1，GND 作为低电平 0。每传输一比特数据 SDA，对应产生一个时钟脉冲 SCL。当 SCL 为高时，SDA 不允许变化；只有在 SCL 为低时，SDA 才可以变化，如图 9-2 所示。

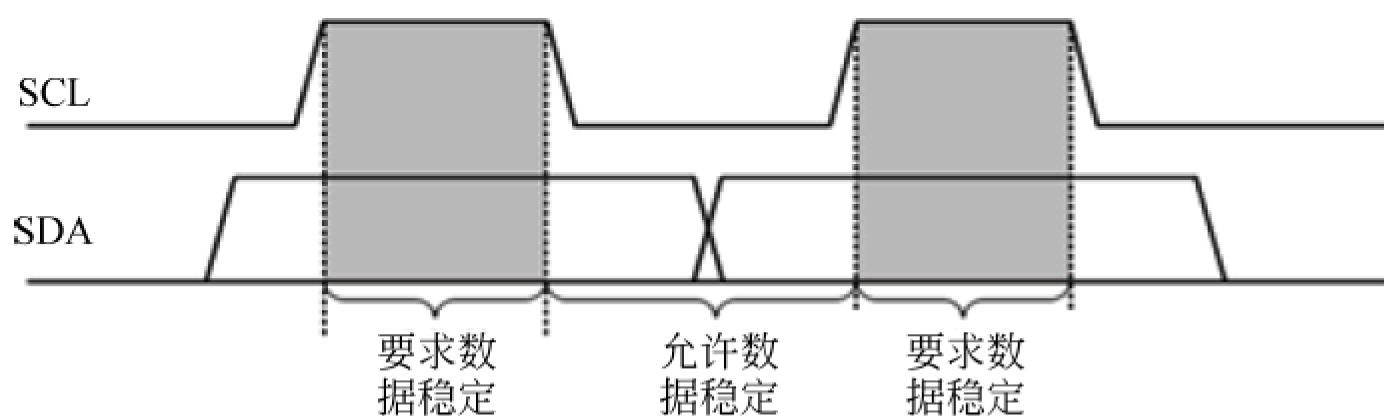


图 9-2　I2C 数据位的传输

2. 开始和结束条件

I2C 的两根线在空闲(IDLE)时均为高电平,SDA 和 SCL 的特定组合表示总线开始或结束一次数据传输,开始和结束的时序如图 9-3 所示。

(1) 开始条件 START(记为 S):SCL 为高时,SDA 由高变低。

(2) 结束条件 STOP(记为 P):SCL 为高时,SDA 由低变高。

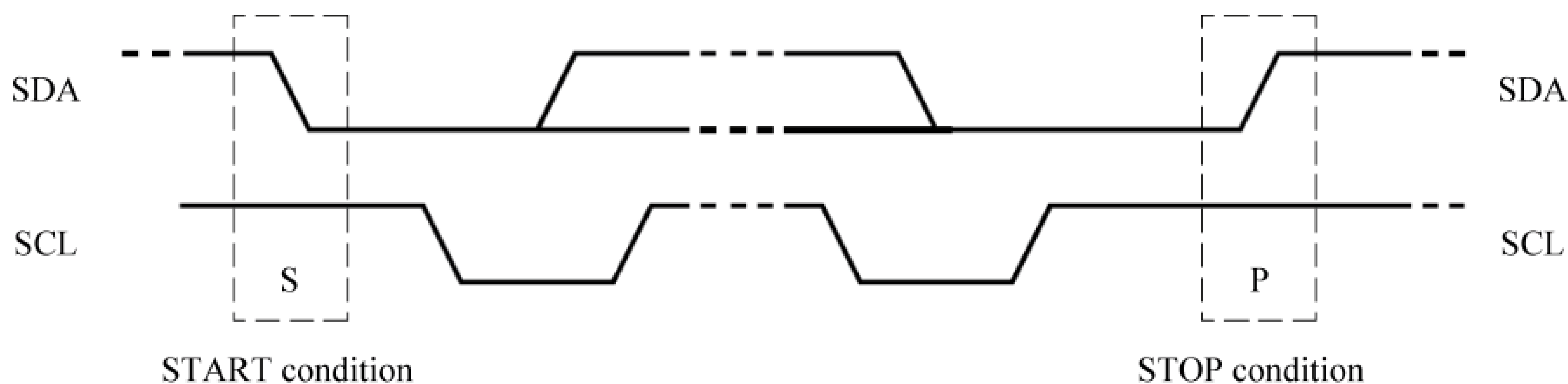


图 9-3　开始和结束条件

开始和结束条件总是由主机(Master)发起的。主机发出开始条件(START)后,总线处于忙的状态;主机发出结束条件(STOP)后,总线处于空闲状态(IDLE)。

在操作中,如果主机发出重复开始条件(Repeated START,记为 Sr)而非结束条件(STOP),则总线仍处于忙的状态。也就是说,重复开始条件(Sr)和开始条件(S)在功能上是相同的。判断开始或结束条件对于具有相应逻辑接口的设备相对简单,对于没有该接口的微控制器,需要在每个 SCL 周期内对 SDA 至少采样 2 次,才能正确检测到开始或结束条件。

3. 字节格式

SDA 上传输字节数据必须是 8 比特长度,每次传输不限定传输的字节数。每个字节(8 位)数据传送完毕后紧接着应答信号(第 9 位,Acknowledge Bit)。数据传输过程中,先发送高位(MSB),再发送低位(LSB),如图 9-4 所示。如果在数据传输过程中,从机如果没有准备好接收或发送下一个字节(比如内部中断需要处理等),它可以通过拉低 SCL 强制主机进入等待状态。直到从机释放 SCL,主机才开始下一个字节的发送或接收。

4. 应答

I2C 协议规定数据传输过程必须包含应答。接收器通过应答位(ACK bit)通知发送的字节已被成功接收,发送器可以进行下一个字节的传输。主机产生传输应答的第 9 个时钟。主机的发送器在应答时钟周期内释放对 SDA 的控制,这样从机接收器可以通过将 SDA 拉

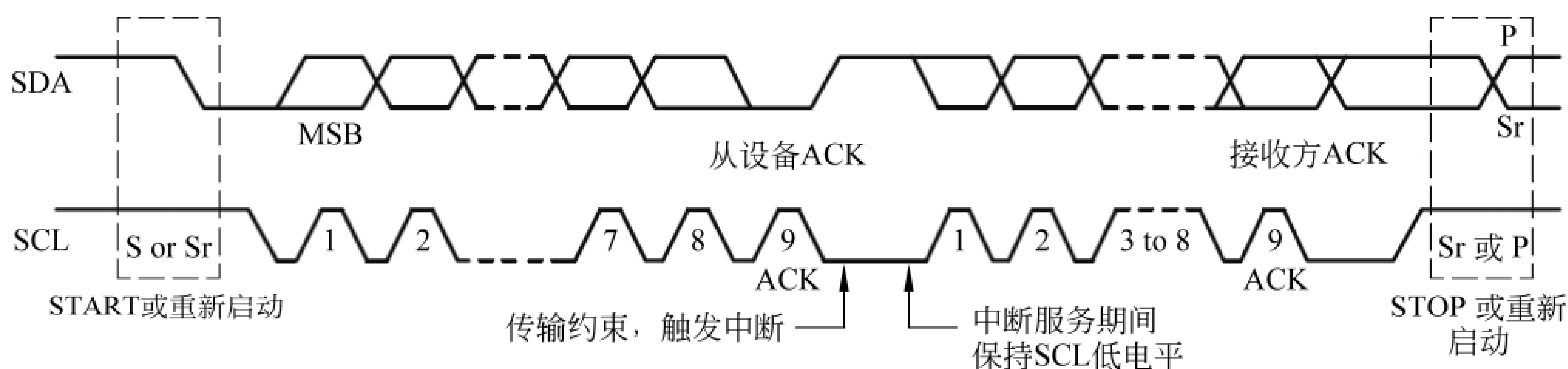


图 9-4　I2C 总线的数据传输

低通知发送器数据已被成功接收，如图 9-5 所示。

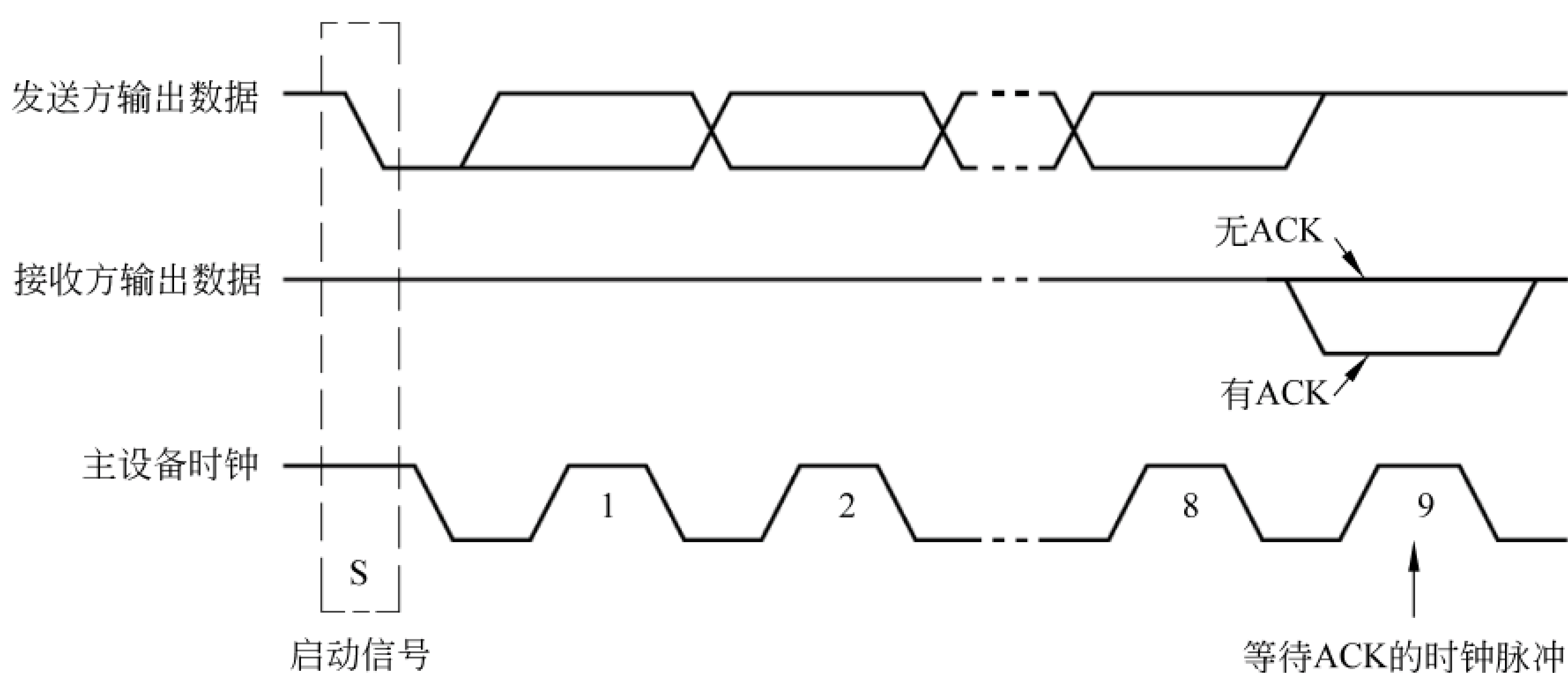


图 9-5　I2C 总线的数据应答

接收器发送 ACK 时，要保证 SCL 为高的同时，SDA 为低电平。如果在第 9 个时钟周期，SDA 为高，表明接收器无应答（NACK），主机可以据此发出结束条件（STOP）命令结束此次传输，或发起重传请求（Repeated START）重新传输数据。以下情况可能导致无应答（NACK）：

- 总线上没有对应地址的接收器件。
- 接收器件没有准备好与主机的通信。
- 接收器件无法解析读取的数据。
- 接收器件无法收取更多的数据。

在连续传输中，主机发送器发送数据，从机接收数据并发送 ACK，主机的接收器在读取了从机发出的最后一个字节数据后，发出 NACK 通知从发送器释放数据线 SDA，主机随后发起结束（STOP）指令完成一次连续数据的传输。

图 9-6 表示主机作为发送器和接收器在写和读情况下的数据格式（ACK/NACK）。

I2C 的一大特点是可以在同一条总线上接多个主机。两个及以上的主机同时发起传输请求时，需要通过某种机制确定哪个主机获得总线的使用权；另外，每个主机都独立产生时钟，时钟速率可能千差万别，这也需要某种机制解决时钟速率不一致的问题。这种机制就是时钟同步（Clock Synchronization）和仲裁（Arbitration）。在单主机的 I2C 系统中，不需要时

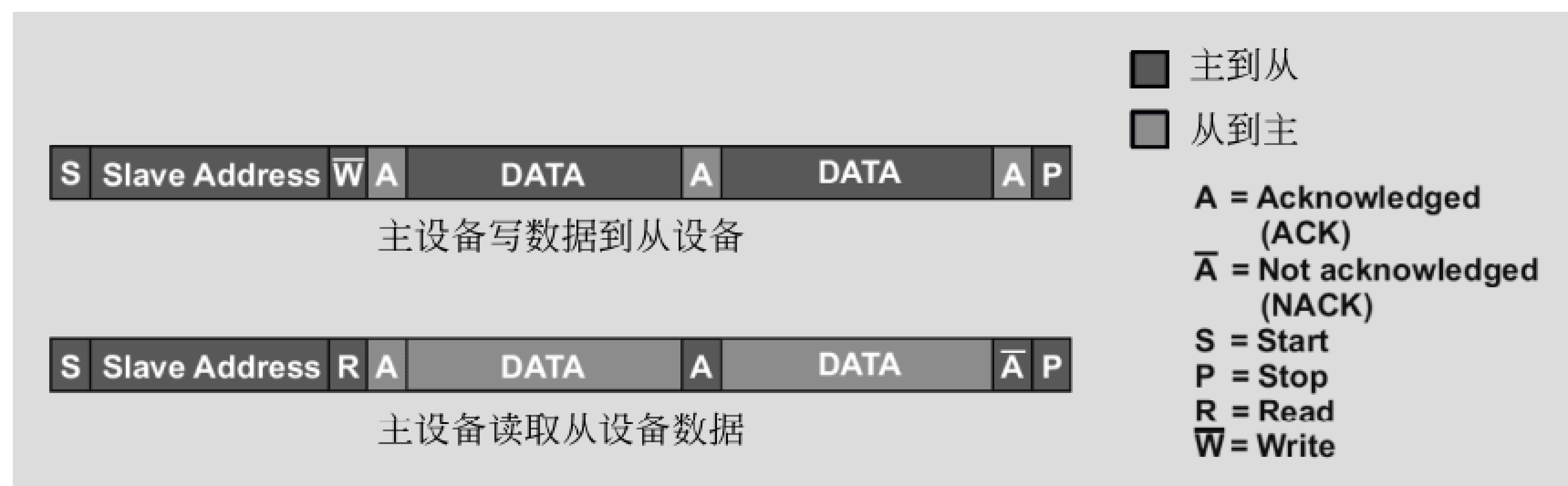


图 9-6　I2C 数据读写的格式

钟同步和仲裁。

5. 地址模式

I2C 的每一个从设备都具有一个设备地址，以便于在总线上连接多个从设备时对从设备进行寻址。I2C 规范有两种地址模式，7 位址模式和 10 位地址模式。

1）7 位地址系统

I2C 总线发送起始信号(START)后，发送的第一个字节由 7 位从设备地址和一位数据方向控制位 R/W 组成，如图 9-7 所示。

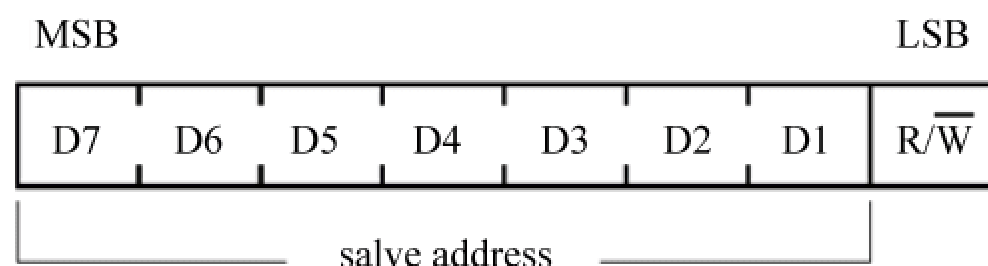


图 9-7　START 后的第一个字节格式

7 位的 I2C 设备地址由类型号和寻址码组成，其中 D7～D4 四位表示器件类型，器件类型是固定的，由 Philip 进行统一管理，D3～D1 是用户自定义的地址码，一般由电路连接不同的高低电平设置，由此可见，同一种类型的设备在 I2C 总线上最多只能连接 8 个。

D0 为数据方向控制位，D0＝1 表示主机读取从机设备，为 0 表示主机向从机发送数据。由读写位的特点可知对于 I2C 总线上的第一个字节，读操作都是奇数，写操作是偶数。

2）7 位地址的连续数据通信

主机发送开始条件后，可以连续进行多个字节的通信，直到主机发送结束条件(P)终止传输。主机也可以通过发起重复开始条件(Sr)进行一次新的传输，而不需要先产生结束条件(P)。

(1) 主机连续向从机发送数据，传输的方向不变。

如图 9-8 所示，主机发起开始条件，然后发送 7 位地址和一个低电平写控制位，立即改为从总线上读取状态，从机处于接收状态，读取总线地址，地址匹配后切换到发送模式向主机发送 ACK 确认，主机收到从机的应答(ACK)后，向从机发送第一个数据 DATA，并等待从机的 ACK 确认。主机发送 NACK，再发送结束条件(P)，结束本次传输。

(2) 主机连续读取从机数据，传输方向不变。

如图 9-9 所示，主机发起开始条件，然后发送 7 位地址和一个高电平读控制位，立即改

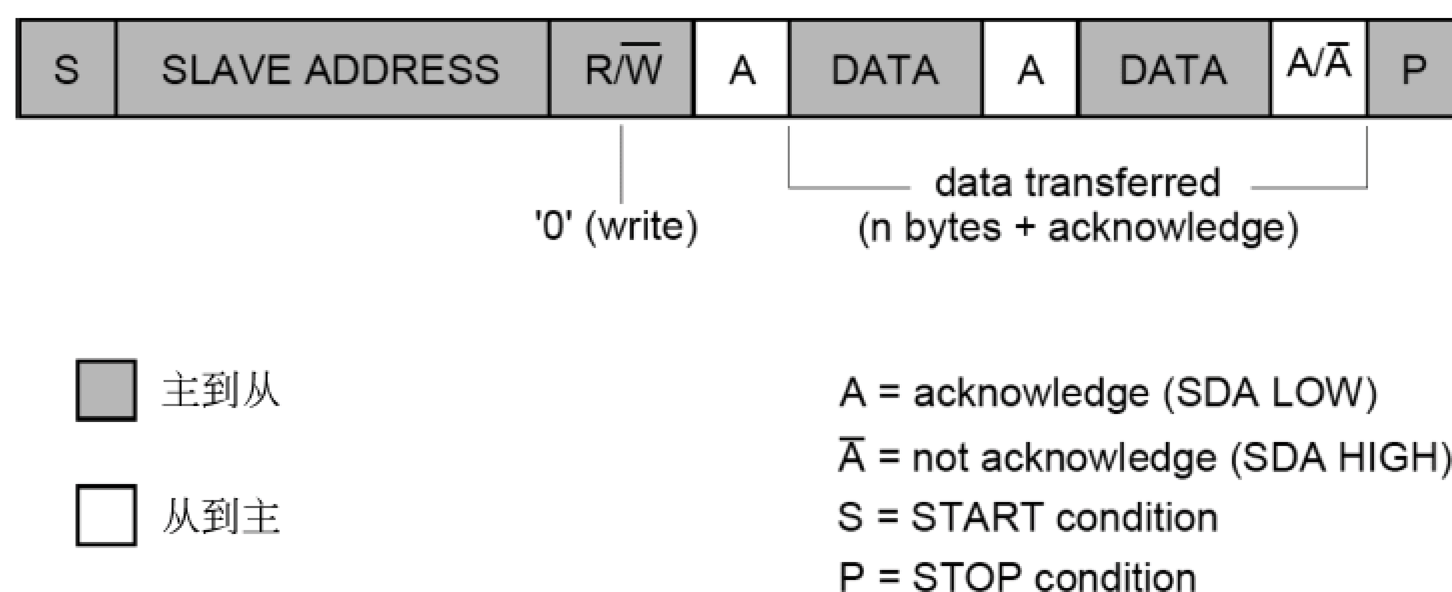


图 9-8　7 位地址主发从收模式

为从总线上读取状态，从机地址匹配后切换到发送模式向主机发送 ACK 确认，并开始发送第一个数据 DATA，主机收到从机的数据后，向从机发送 ACK 确认，并切换到接收模式接收从机的下一个数据。第一个 ACK 由从机发出，此后的 ACK 由主机发出。主机发送 NACK，再发送结束条件(P)，结束本次传输。

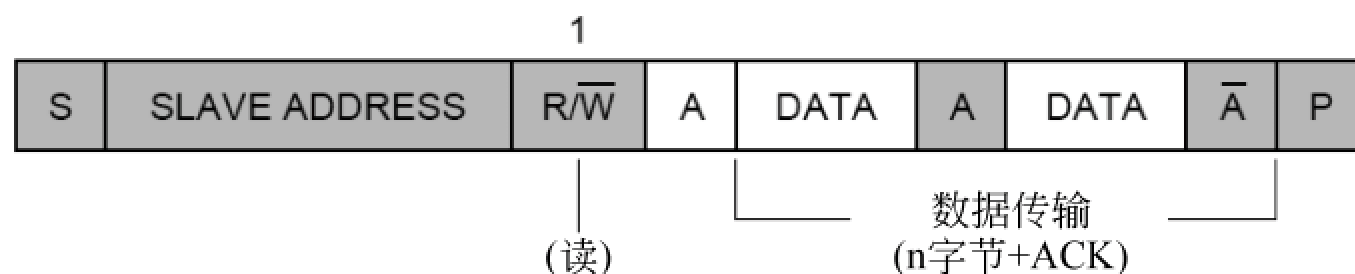


图 9-9　主机在第一个字节后立即读取从机内容

(3) 读写混合模式，传输方向改变。

如图 9-10 所示，混合模式中，若要改变传输方向，则需要重新发送起始条件和从设备地址及读写控制位。

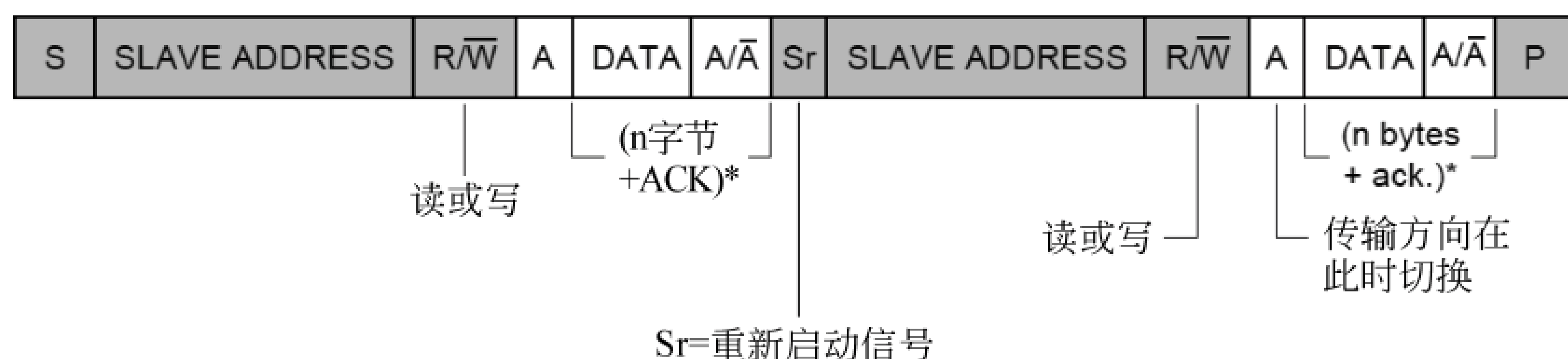


图 9-10　混合模式

3) 10 位地址模式

采用 10 位地址系统扩充了 I2C 系统的地址范围。7 位和 10 位地址设备可以共存于同一个 I2C 总线系统，并且可工作在所有速度模式。目前使用 10 位地址系统的 I2C 设备不多。

10 位从机地址由两个字节 16 位组成，如图 9-11 所示，地址表示为 11110xx xxxxxxxx，发送地址时，先发送高 7 位(包含 10 位地址的最高两位)bits[15：9]，bit[8]用作读写方向控制位表明传输方向，此时总线上的所有从设备对比总线上第一个字节的前七位(1111 0XX)是否和自身地址一致，可能有一个以上设备会检测到地址匹配(因为只对比了 10 位地址的

最高 2 位),它们都会产生响应 A1。收到 A1 后,主机发出 10 位地址的低 8 位(注意此次地址不包含读写方向控制位)bits[7:0],所有上面响应的从机对比总线上第二个字节和它们各自地址的后八位(XXXX XXXX)是否一致。只有一个设备的地址匹配,并产生响应 A2。被寻址的从机一直受主机控制,直到 STOP 或 Sr 指向另外的地址。



图 9-11　主发送器寻址从接收器(10 位地址空间)

在混合传输下,主接收器改变读写方向时,无需每次都发送两次地址,如图 9-12 所示,在发送起始条件和 10 位地址之后,若需要改变读写方向,只需重新发送起始条件和第一个 7bit 地址及传输方向控制位即可,从机一直占用总线,直到接收到 STOP 或 Sr 指向另一个从机地址。

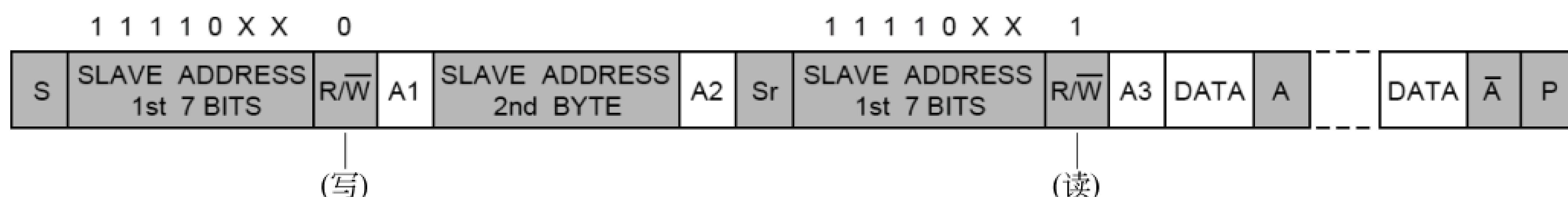


图 9-12　主接收器寻址从发送器(10 位地址空间)

从机地址中,0000000 0(R/W)表示通用广播地址,0000000 1(R/W)用于为不带 I2C 控制器的微处理器采用软件方式检测启动条件。

9.3　STM32L152 I2C 总线控制器

STM32L152 内部集成了两个独立的 I2C 总线控制器,I2C 总线控制器控制所有 I2C 总线特定的时序、协议、仲裁和定时,支持 CRC 码的生成和校验、系统管理总线 SMBus 和电源管理总线 PMBus 以及 DMA 传输,其主要特点为:

- 多主机功能:该模块既可做主设备也可做从设备。
- 可响应 2 个从地址的双地址能力。
- 产生和检测 7 位/10 位地址和广播呼叫。
- 支持标准速度(100kHz)和快速(高达 400kHz)两种模式。
- 支持多种状态标志和错误标志便于程序控制总线状态。
- 支持 2 个中断向量,用于地址/数据通信成功和错误处理。
- 可选的拉长时钟功能。
- 可配置的 PEC(信息包错误检测)的产生或校验。

• 兼容 SMBus 2.0。

I2C 控制器接收和发送数据，并将数据从串行转换成并行，或将并行转换成串行，接口通过数据引脚(SDA)和时钟引脚(SCL)连接，其内部结构如图 9-13 所示。

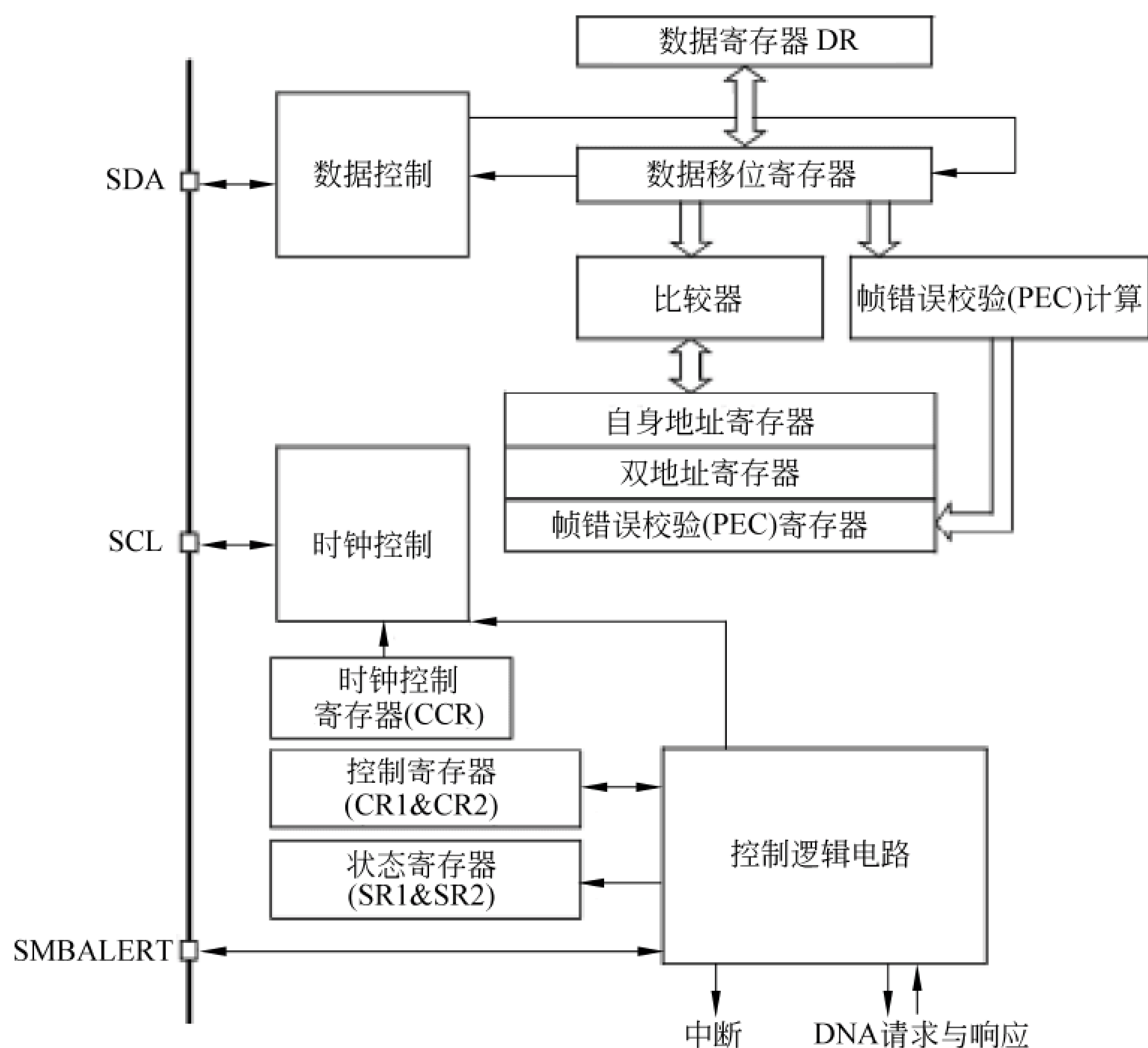


图 9-13　I2C 控制器内部结构

数据寄存器 DR 用于存储将要发送到 SDA 总线上的数据或者从 SDA 总线上读取的数据，数据通过一个移位寄存器进行发送或接收，发送数据时，数据寄存器的值被复制到移位寄存器，移位寄存器将 LSB 先送到总线上，直到 MSB 发送完成。接收时，当一个字节的所有位收到后，移位寄存器的数据被复制到数据寄存器 DR。

数据控制逻辑用于控制 SDA 的收发方向和 ACK 确认。比较器用于对从模式自身地址和主设备寻址发送的地址进行匹配，PEC 计算部分执行校验算法并和总线上的 PEC 数据进行比对。

控制逻辑电路通过时钟控制寄存器、控制寄存器对 I2C 的工作模式，时钟进行配置，维护状态寄存器，并向 CPU 发送中断和 DMA 请求。

I2C 控制器可以配置为下述 4 种模式中的一种运行：

• 从发送器模式。
• 从接收器模式。
• 主发送器模式。
• 主接收器模式。

MCU 启动后，I2C 默认地工作于从模式。I2C 控制器在接收到生成起始条件命令配置后自动地从从模式切换到主模式；当仲裁丢失或产生停止信号时，则从主模式切换到从模式。主模式时，I2C 接口启动数据传输并产生时钟信号。串行数据传输总是以起始条件开始并以停止条件结束。起始条件和停止条件都是在主模式下由软件控制产生。

从模式时，I2C 接口能识别它自己的地址（7 位或 10 位）和广播呼叫地址。软件能够控制开启或禁止广播呼叫地址的识别。

数据和地址按 8 位/字节进行传输，高位在前。跟在起始条件后的 1 或 2 个字节是地址（7 位模式为 1 个字节，10 位模式为 2 个字节）。地址只在主模式发送。在一个字节传输的 8 个时钟后的第 9 个时钟期间，接收器必须回送一个应答位（ACK）给发送器。软件可以开启或禁止应答（ACK），并可以设置 I2C 接口的地址（7 位、10 位地址或广播呼叫地址）。

9.4　I2C 寄存器描述

I2C 控制器的寄存器如表 9-1 所示。

表 9-1　I2C 控制器寄存器列表

寄存器名称	偏移量	功　　能	复　位　值
控制寄存器 1 I2C_CR1	0x00	起始信号、停止信号、使能等控制	0x0000 0000
控制寄存器 2 I2C_CR2	0x04	中断、DMA 和频率控制	0x0000 0000
自身地址寄存器 1 I2C_OAR1	0x08	从模式时的地址	0x0000 0000
自身地址寄存器 2 I2C_OAR2	0x0C	从模式时第二个地址	0x0000 0000
数据寄存器 I2C_DR	0x10	发送和接收数据	0x0000 0000
状态寄存器 1 I2C_SR1	0x14	接收、发送、地址匹配及错误状态	0x0000 0000
状态寄存器 1 I2C_SR2	0x18	主从、发送方向和忙状态指示	0x0000 0000
时钟控制寄存器 I2C_CCR	0x1C	SCLK 参数配置	0x0000 0000
Trise 寄存器 I2C_TRISE	0x20	电平边沿变化速度	0x0000 0000

1. 控制寄存器 1（I2C_CR1）

控制寄存器的有效域定义如图 9-14 所示。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SWRST	保留	ALERT	PEC	POS	ACK	STOP	START	NO STRETCH	ENGC	ENPEC	ENARP	SMB TYPE	保留	SMBUS	PE
rw	res	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	res	rw	rw

图 9-14　控制寄存器 CR1

SWRST（Software Reset）：软件复位，该位被置 1 时，I2C 处于复位状态。

ALERT：SMBus 提醒，用于 SMBus 总线。

PEC(Packet Error Checking)：数据包出错检测。

POS(Position)：应答 ACK 的发送时机和 PEC 位置指示(用于数据接收)。

ACK：应答使能，0 表示无应答返回，1 表示在接收到一个字节后返回一个应答。

STOP：停止条件产生，该位置 1 表示产生一个停止条件，主模式下，在当前字节传输或在当前起始条件发出后产生停止条件，从模式下，在当前字节传输或释放 SCL 和 SDA 线后产生停止条件。该位可由软件设置，当检测到停止条件或超时错误时，硬件将自动清除该位。

START：起始条件产生，该位置 1 表示产生一个起始条件，主模式下设置该位为 1 将重复产生起始条件，从模式下，当总线空闲时产生起始条件。该位可由软件设置，当起始条件发出后该位由硬件自动清除。

NOSTRETCH：禁止时钟延长(从模式)，该位用于当 ADDR 或 BTF 标志被置位，在从模式下禁止时钟延长，直到它被软件复位。

ENGC(General Call Enable)：广播呼叫使能，置 0 表示禁止广播呼叫。以非应答响应地址 00h，置 1 表示允许广播呼叫．以应答响应地址 00h。

ENPEC(PEC Enable)：PEC 使能，置 0 禁止 PEC 计算，置 1 开启 PEC 计算。

ENARP(ARP Enable)：ARP 使能 置 0 禁止 ARP，置 1 使能 ARP。ARP 是 SMBus 功能。

SMBTYPE(SMBus Type)：SMBus 类型 0 表示 SMBus 设备，1 表示 SMBus 主机。

SMBUS(SMBus Mode)：SMBus 模式，置 1 表示启用 SMBus 模式，置 0 表示 I2C 模式。

PE(Peripheral enable)：I2C 模块使能，置 1 表示启用 I2C 模块。

2. 控制寄存器 2(I2C_CR2)

控制寄存器 2 的有效域定义如图 9-15 所示。

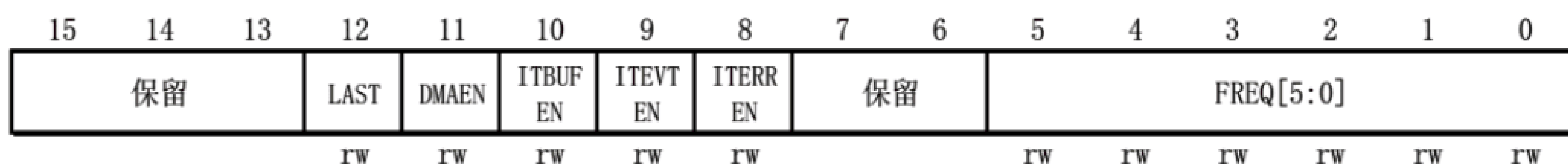


图 9-15 控制寄存器 CR2

LAST：DMA 最后一次传输，置 1 表示下一次 DMA 的 EOT 是最后的传输。

DMAEN(DMA Request Enable)：DMA 请求使能，0 表示禁止 DMA 请求，1 表示当 TxE=1 或 RxNE =1 时，允许 DMA 请求。

ITBUFEN(Buffer Interrupt Enable)：缓冲器中断使能，0 表示当 TxE=1 或 RxNE=1 时，不产生任何中断，1 表示当 TxE=1 或 RxNE=1 时，产生事件中断。

ITEVTEN(Event Interrupt Enable)：事件中断使能，置 0 禁止事件中断，置 1 允许事件中断。在下列条件下，将产生该中断：

- SB=1(主模式)；
- ADD10=1(主模式)；
- 如果 ITBUFEN=1，TxE 事件为 1；
- ADDR=1(主/从模式)
- STOPF=1(从模式)
- 如果 ITBUFEN=1，RxNE 事件为 1

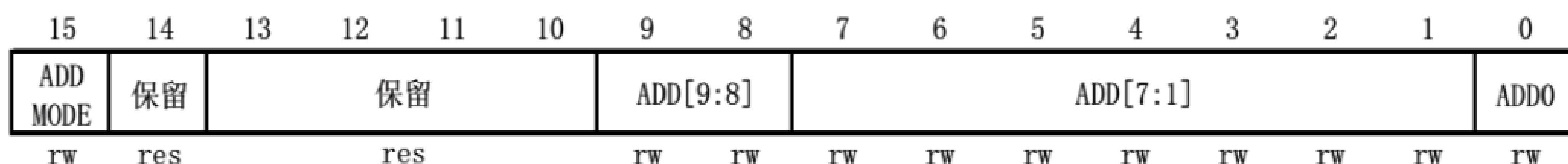
－BTF=1,但是没有 TxE 或 RxNE 事件;

ITERREN(Error Interrupt Enable):出错中断使能 0 表示禁止出错中断,1 表示允许出错中断。如果 BERR＝1、ARLO＝1、AF＝1、OVR＝1、PECERR＝1、TIMEOUT、SMBAlert＝1,则产生中断。

FREQ[5:0]:I2C 模块时钟频率,时钟频率允许的范围为 2M～50MHz,但不能超过总线时钟。FREQ 取 000000 和大于 110010 的值表示禁用,000010～110010 分别表示 2MHz,3MHz,…,50MHz。

3. 自身地址寄存器 1(I2C_OAR1)

自身地址寄存器 1 的有效域定义如图 9-16 所示。



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD MODE	保留	保留				ADD[9:8]		ADD[7:1]							ADD0
rw	res	res				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 9-16　自身地址寄存器 OAR1

ADDMODE(Address Mode):从模式下寻址模式,0 表示 7 位地址,1 表示 10 位地址。

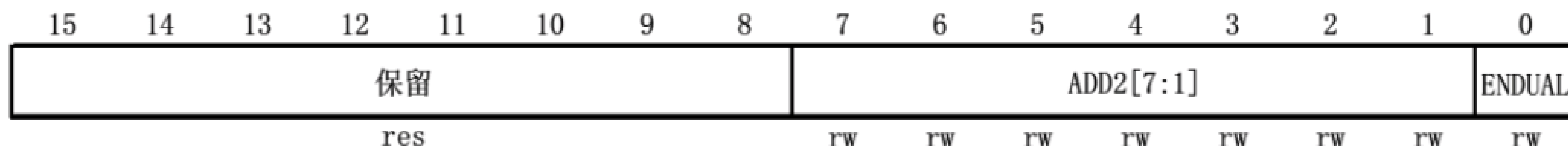
ADD[9:8]:从设备地址 9～8 位:7 位地址模式时无效,10 位地址时为地址的 9～8 位。

ADD[7:1]:从设备地址 7～1 位。

ADD0:从设备地址 0 位,7 位地址模式时用于表示读写方向位,10 位地址模式时为地址第 0 位。

4. 自身地址寄存器 2(I2C_OAR2)

自身地址寄存器 2 的有效域定义如图 9-17 所示。



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留								ADD2[7:1]							ENDUAL
res								rw	rw	rw	rw	rw	rw	rw	rw

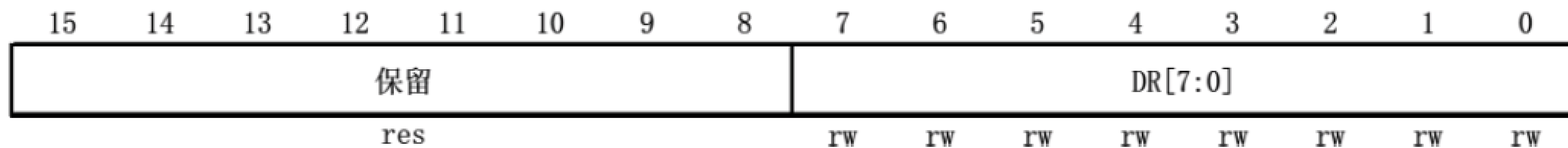
图 9-17　自身地址寄存器 OAR2

ADD2[7:1]:从设备地址 7～1 位。

ENDUAL(Dual Address Mode Enable):双地址模式使能位,置 0 表示在 7 位地址模式下,只有 OAR1 被识别,置 1 表示在 7 位地址模式下,OAR1 和 OAR2 都被识别。

5. 数据寄存器(I2C_DR)

数据寄存器有效域定义如图 9-18 所示。



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留								DR[7:0]							
res								rw	rw	rw	rw	rw	rw	rw	rw

图 9-18　数据寄存器 DR

DR[7:0]:8 位数据寄存器,用于存放接收到的数据或放置用于发送到总线的数据,发送器模式:当写一个字节至 DR 寄存器时,自动启动数据传输。一旦传输开始(TxE＝1),

如果能及时把下一个需传输的数据写入 DR 寄存器，I2C 模块将保持连续的数据流。接收器模式：接收到的字节被复制到 DR 寄存器(RxNE=1)。在接收到下一个字节(RxNE=1)之前读出数据寄存器，即可实现连续的数据传送。在从模式下，地址不会被复制进数据寄存器 DR，且硬件不管理写冲突，如果 TxE=0，仍能写入数据寄存器，此时数据会发生错误。

6. 状态寄存器 1(I2C_SR1)

状态寄存器 1 的有效域定义如图 9-19 所示。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SMB ALERT	TIME OUT	保留	PEC ERR	OVR	AF	ARLO	BERR	TxE	RxNE	保留	STOPF	ADD10	BTF	ADDR	SB
rc w0	rc w0	res	rc w0	rc w0	rc w0	rc w0	rc w0	r	r	res	r	r	r	r	r

图 9-19 状态寄存器 SR1

SMBALERT (SMBusAlert)：SMBus 告警提醒。

TIMEOUT：超时或 Tlow 错误，该位为 1 表明 SCL 处于低已达到 25ms(超时)，或者主机低电平累积时钟扩展时间超 10ms(Tlow：mext)，或从设备低电平累积时钟扩展时间超过 25ms。从模式下该位为 1 时从设备复位，硬件释放总线，主模式下设置该位，硬件发出停止条件。该位可由软件写 0 清除，或在 PE=0 时由硬件自动清除。

PECERR(PEC ERROR)：在接收时发生 PEC 错误。0 表示无 PEC 错误，接收到 PEC 后接收器返回 ACK(如果 ACK=1)；1 表示有 PEC 错误，接收到 PEC 后接收器返回 NACK(不管 ACK 是什么值)。该位可由软件写 0 清除，或在 PE=0 时由硬件自动清除。

OVR：过载/欠载标志，1 表示出现出现过载/欠载。当 NOSTRETCH=1，在接收模式中当收到一个新的字节时(包括 ACK 应答脉冲)，数据寄存器里的内容还未被读出，则新接收的字节将丢失；在发送模式中当要发送一个新的字节时，却没有新的数据写入数据寄存器，同样的字节将被发送两次，在这两种情况下，该位被硬件置自动置 1。该位可由软件写 0 清除，或在 PE=0 时由硬件自动清除。

AF(Acknowledge Failure)：应答失败，该位为 1 表示应答失败。当没有返回应答 ACK 时，硬件将该位自动置为 1。该位可由软件写 0 清除，或在 PE=0 时由硬件自动清除。

ARLO(Arbitration Lost)：仲裁丢失(主模式)，该位为 1 表明检测到仲裁丢失。当接口失去对总线的控制给另一个主机时，硬件将置该位为 1。该位可由软件写 0 清除，或在 PE=0 时由硬件自动清除。在 ARLO 事件之后，I2C 接口自动切换回从模式(M/SL=0)。

BERR(Bus Error)：总线出错，该位为 1 表示起始条件或停止条件出错，当 IIC 总线检测到错误的起始或停止条件，硬件将该位置 1。该位可由软件写 0 清除，或在 PE=0 时由硬件自动清除。

TxE(Transmit Data Register Empty)：数据发送寄存器为空标志，0 表示非空，1 表示空。在发送数据时，数据寄存器为空时该位被置 1，在发送地址阶段不设置该位。软件写数据到 DR 寄存器可清除该位；或者在发生一个起始或停止条件后、PE=0 时由硬件自动清除。如果收到一个 NACK，或下一个要发送的字节是 PEC(PEC=1)，该位不被置位。

RxNE(Receive Data Register Empty)：接收数据寄存器非空标志，1 表示非空，0 表示

空。在接收时,当数据寄存器不为空时该位被置1。在接收地址阶段,该位不被置位。软件对数据寄存器的读写操作清除该位,或当PE=0时由硬件清除。

STOPF(Stop Detection Flag):从模式停止条件检测位,该位置1表示检测到停止条件。在一个应答之后(如果ACK=1),当从设备在总线上检测到停止条件时,硬件将该位置1。软件读取SR1寄存器后,对CR1寄存器的写操作将清除该位,或当PE=0时,硬件清除该位。在收到NACK后,STOPF位不被置位。

ADD10:主模式10位地址序列发送标志,该位置1表示主设备已经将从设备地址的第一个字节发送出去。在10位地址模式下,当主设备已经将地址的第一个字节发送出去时,硬件将该位置1。软件读取SR1寄存器后,对CR1寄存器的写操作将清除该位,或当PE=0时,硬件清除该位。收到一个NACK后,ADD10位不被置位。

BTF(Byte Transfer Finished):字节发送结束。0表示字节发送未完成,1表示字节发送结束。当NOSTRETCH=0时,在下列情况下硬件将该位置1:

- 当收到一个新字节(包括ACK脉冲)且数据寄存器还未被读取(RxNE=1)。
- 当一个新数据将被发送且数据寄存器还未被写入新的数据(TxE=1)。

软件读取SR1寄存器后,对数据寄存器的读或写操作将清除该位;或在传输中发送一个起始或停止条件后,或当PE=0时,由硬件清除该位。在收到一个NACK后,BTF位不会被置位。

ADDR:主模式地址发送标记/从模式地址匹配标记。从模式中,该位置0表示地址不匹配或没有收到地址,当收到的地址与OAR寄存器中的地址匹配时硬件自动将该位置1;主模式中,该位为0表示地址发送没有结束,1表示地址发送结束。10位地址模式时,当收到地址的第二个字节的ACK后该位被置1;7位地址模式时,当收到地址的ACK后该位被置1。在软件读取SR1寄存器后,对SR2寄存器的读操作将清除该位,或当PE=0时,由硬件清除该位。

SB(Start Bit):主模式起始位标志,0表示未发送起始条件,1表示起始条件已发送。当发送出起始条件时该位被置1。软件读取SR1寄存器后,写数据寄存器的操作将清除该位,或当PE=0时,硬件清除该位。

7. 状态寄存器2(I2C_SR2)

状态寄存器2的有效域定义如图9-20所示。

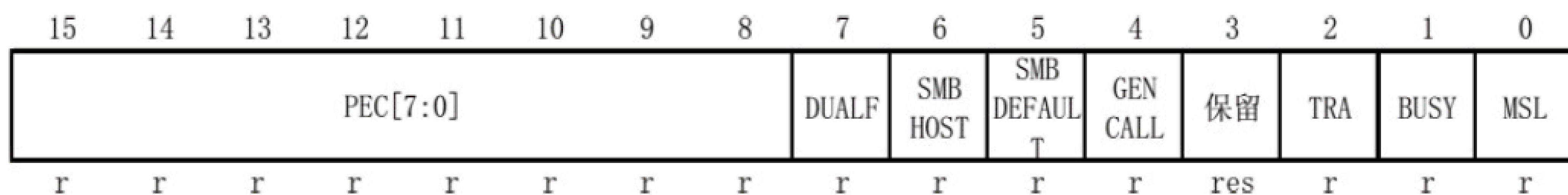


图9-20　状态寄存器SR2

PEC[7:0](Packet Error Checking):数据包出错检测,当ENPEC=1时,PEC[7:0]存放内部的PEC的值。

DUALF(Dual Flag):从模式双地址标志,该位为0表示接收到的地址与OAR1内的

内容相匹配,1 表示接收到的地址与 OAR2 内的内容相匹配。在产生一个停止条件或一个重复的起始条件时,或 PE=0 时,硬件将该位清除。

SMBHOST:SMBus 主机地址标志。

SMB DEFAULT:SMBus 默认地址标志。

GENCALL:广播呼叫地址标志。

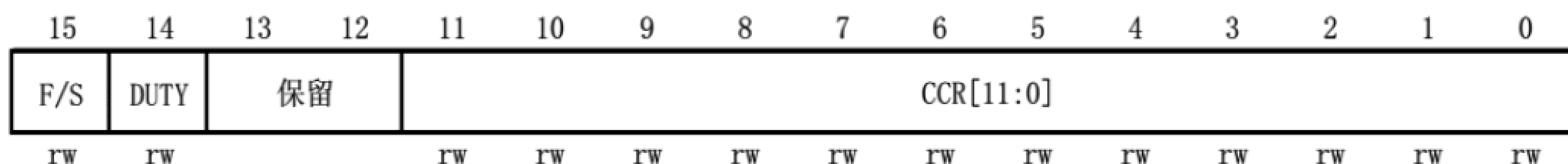
TRA(Transmit/Receive):发送/接收标志,0 表示接收到数据,1 表示数据已发送。该位根据地址字节的 R/W 位来设定。在检测到停止条件(STOPF=1)、重复的起始条件或总线仲裁丢失(ARLO=1)后,或当 PE=0 时,硬件将其清除。

BUSY:总线忙标志,0 表示在总线上无数据通信,1 表示总线上正在进行数据通信。在检测到 SDA 或 SCl 为低电平时,硬件将该位置 1,当检测到停止条件时,硬件将该位清除。

MSL(Master Slave):主从模式,0 表示从模式,1 表示主模式,当总线配置为主模式(SB=1)时,硬件将该位置位;当总线上检测到一个停止条件、仲裁丢失(ARLO=1 时)或当 PE=0 时,硬件清除该位。

8. 时钟控制寄存器(I2C_CCR)

时钟控制寄存器的有效域定义如图 9-21 所示。



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
F/S	DUTY	保留		CCR[11:0]											
rw	rw			rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 9-21　时钟控制寄存器 CCR

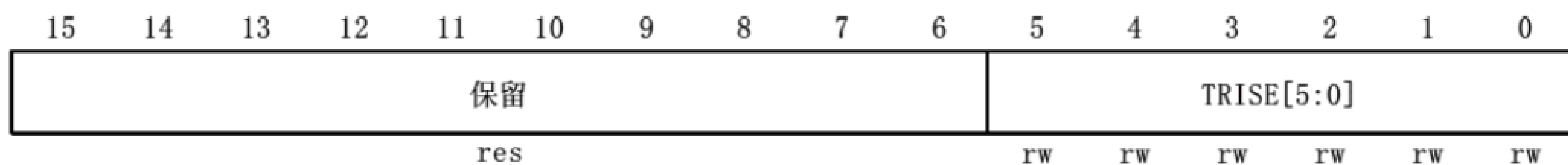
F/S(Fast/Standard Mode Selection):I2C 主模式选项,0 表示标准模式,1 表示快速模式。

DUTY:快速模式时的占空比,0 表示快速模式下 Tlow/Thigh = 2,1 表示快速模式下 Tlow/Thigh = 16/9。

CCR[11:0]:时钟系数,用于设置主模式下的 SCL 时钟。在 I2C 标准模式或 SMBus 模式下:T_{high}= CCR×T_{PCLK1},T_{low}= CCR ×T_{PCLK1};在 I2C 快速模式下,如果 DUTY = 0,T_{high}= CCR ×T_{PCLK1},T_{low}= 2×CCR×T_{PCLK1};如果 DUTY = 1,T_{high}= 9×CCR×T_{PCLK1},T_{low}= 16×CCR×T_{PCLK1}。CCR 允许设定的最小值为 0x04,在快速 DUTY 模式下允许的最小值为 0x01;只有在关闭 I2C 时(PE = 0)才能设置 CCR 寄存器,且 F_{pclk}应当是 10MHz 的整数倍,这样可以正确产生 400kHz 的快速时钟。

9. TRISE 寄存器(I2C_TRISE)

上升时间寄存器的有效域定义如图 9-22 所示。



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留										TRISE[5:0]					
res										rw	rw	rw	rw	rw	rw

图 9-22　Trise 寄存器

TRISE[5：0](Time of Rise)：在快速/标准模式下的最大上升时间，这些位必须设置为I2C总线规范里给出的最大的SCL上升时间，增长步幅为1。例如：标准模式中最大允许SCL上升时间为1000ns。如果在I2C_CR2寄存器中FREQ[5：0]中的值等于0x08，则TPCLK1 =125ns，故TRISE[5：0]中必须写入09h(1000ns/125 ns = 8+1)。只有当I2C被禁用(PE=0)时，才能设置TRISE[5：0]。

9.5　I2C数据通信流程

9.5.1　I2C从模式通信

默认情况下，I2C接口总是工作在从模式。由从模式切换到主模式，需要产生一个起始条件。为了产生正确的时序，必须在I2C_CR2寄存器中设定该模块的输入时钟。输入时钟的频率在标准模式下至少是2MHz，快速模式下至少4MHz。

一旦检测到起始条件，I2C SDA线上接收到的地址被送到移位寄存器。然后与芯片自己的地址OAR1和OAR2(当ENDUAL=1时)或者广播呼叫地址(当ENGC=1时)相比较。

如果头段或地址不匹配，I2C将其忽略并等待另一个起始条件。如果地址匹配，I2C接口产生以下时序：

- 若启用ACK(寄存器I2C_CR1的ACK=1)，则产生一个应答脉冲。
- 硬件自动设置I2C_SR1寄存器的ADDR位为1，如果I2C_CR2中配置了ITEVFEN=1，则产生一个中断。
- 如果I2C_OAR2寄存器的ENDUAL位为1，软件必须读I2C_SR2寄存器的DUALF位，以确认响应了哪个从地址。

在从模式下I2C_SR2的TRA位指示当前是处于接收器模式还是发送器模式。

1. 从模式下的发送

从模式下的发送指的是接收主设备的读请求，从设备向主设备主动发送数据。从模式下的发送流程如图9-23所示，图中EVx表示I2C通信过程中产生的事件，若设置了ITEVFEN=1，则会产生一个中断。

图9-23　从模式发送时序

从发送器在接收到正确的设备地址，发送应答脉冲 A(启用 ACK)后，I2C 控制器产生事件 EV1，ADDR1 被置 1，读 I2C_SR1 寄存器然后再读 I2C_SR2 寄存器可清除 ADDR 位。

若此时移位寄存器和数据寄存器中均没有数据，TxE=1，此时产生事件 EV3_1，可将数据写入数据寄存器 DR。

写入数据寄存器 DR 后，TxE=0，此时移位寄存器为空，则数据立即从数据寄存器写入到移位寄存器，移位寄存器非空，TxE=1，产生事件 EV3；如果接收设备地址 R/W 方向控制位为 0，则从发送器将字节从 DR 寄存器经由内部移位寄存器发送到 SDA 线上。

一个数据发送完成后，从发送器接收主设备的应答脉冲(启用 ACK)，若收到应答脉冲，TxE 位被硬件置 1，如果设置了 ITEVFEN 和 ITBUFEN 位，则产生一个中断。如果 TxE 位被置 1，且收到了 NACK(AF=1)，产生事件 EV3_2，则从设备不会再向主设备发送数据，在下一个数据发送结束之前没有新数据写入到 I2C_DR 寄存器，则 BTF 位被置 1，在清除 BTF 之前 I2C 接口将保持 SCL 为低电平；读出 I2C_SR1 之后再写入 I2C_DR 寄存器将清除 BTF 位。

2. 从模式下的接收

从模式下的接收指的是主设备向从设备写数据，从设备不向主设备发送数据。从模式接收的流程如图 9-24 所示。在接收到地址(产生事件 EV1)并清除 ADDR 后，从接收器将通过内部移位寄存器从 SDA 线接收到的字节存进 DR 寄存器。I2C 接口在接收到每个字节后都执行下列操作：

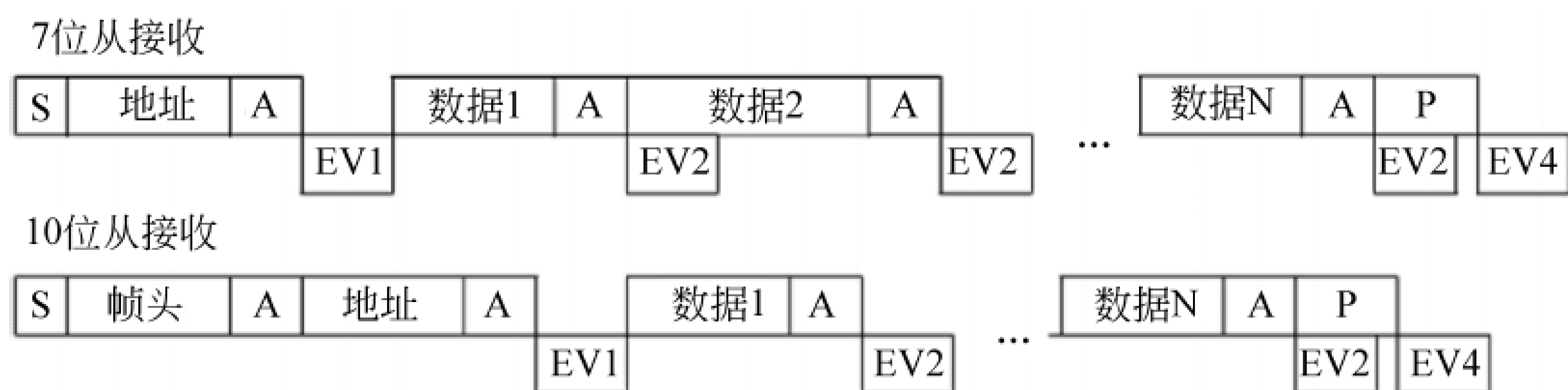


图 9-24 从模式接收时序

- 如果设置了 ACK 位，则产生一个应答脉冲。
- 硬件设置 RxNE=1(产生事件 EV2)，如果设置了 ITEVFEN 和 ITBUFEN 位，则产生一个中断。

如果 RxNE 被置位，并且在接收新的数据结束之前 DR 寄存器未被读出，BTF 位被置位，在清除 BTF 之前 I2C 接口将保持 SCL 为低电平；读出 I2C_SR1 之后再写入 I2C_DR 寄存器将清除 BTF 位。

在传输完最后一个数据字节后，主设备产生一个停止条件，I2C 接口检测到这一条件时：设置 STOPF=1，产生事件 EV4，如果设置了 ITEVFEN 位，则产生一个中断。读 SR1 寄存器，再写 CR1 寄存器可清除 STOPF 位。

9.5.2 I2C 主模式通信

在主模式时，I2C 接口启动数据传输并产生时钟信号。串行数据传输总是以起始条件开始并以停止条件结束。当通过 START 位在总线上产生了起始条件，设备就进入了主模式。以下是主模式的操作顺序：

(1) 在 I2C_CR2 寄存器中设定该模块的输入时钟以产生正确的时序，输入时钟在标准模式下至少为 2MHz，快速模式下至少为 4MHz。

(2) 配置时钟控制寄存器。

(3) 配置上升时间寄存器。

(4) 编程 I2C_CR1 寄存器启动外设。

(5) 置 I2C_CR1 寄存器中的 START 位为 1，产生开始条件。

当 BUSY＝0 时，设置 START＝1，I2C 接口将产生一个开始条件并切换至主模式(M/SL 位置位)。在主模式下，设置 START＝1 将在当前字节传输完后由硬件产生一个重开始条件。

一旦发出开始条件，SB 位被硬件置为 1，产生事件 EV5，如果设置了 ITEVFEN 位，则产生一个中断。然后主设备写 DR 寄存器(从设备地址)清除 SB 位，等待读 SR1 寄存器。

(6) 在 10 位地址模式时，先发送一个头段序列，若收到一个响应 ACK，则 ADD10 位被硬件置位，产生事件 9，如果设置了 ITEVFEN 位，则产生一个中断。读 SR1 寄存器，再将第二个地址字节写入 DR 寄存器(清除 ADD10)，若收到一个响应 ACK，则 ADDR 位被硬件置位，产生事件 EV6，如果设置了 ITEVFEN 位，则产生一个中断。主设备等待一次读 SR1 寄存器和读 SR2 寄存器清除 ADDR 位。

(7) 在 7 位地址模式时，只需送出一个地址字节。一旦该地址字节被送出，收到 ACK 响应，ADDR 位被硬件置为 1，产生事件 EV6，如果设置了 ITEVFEN 位，则产生一个中断。主设备等待一次读 SR1 寄存器和一次读 SR2 寄存器清除 ADDR 位。

根据送出从地址的最低位，主设备决定进入发送器模式还是进入接收器模式。在 7 位地址模式时，要进入发送器模式，主设备发送从地址时置最低位为 0，要进入接收器模式，主设备发送从地址时置最低位为 1。

在 10 位地址模式时，要进入发送器模式，主设备先送头字节 11110xx0(xx 代表 10 位地址中的最高 2 位)，再发剩余的 8 位地址；要进入接收器模式，主设备先送头字节(11110xx0)，再发剩余的 8 位地址，然后再重新发送一个开始条件，后面跟着头字节(11110xx1)。

TRA 位指示主设备是在接收器模式还是发送器模式。

1. 主模式下的发送

主模式下的发送时序如图 9-25 所示，在发送了地址和清除了 ADDR 位后，主设备通过内部移位寄存器将字节从 DR 寄存器发送到 SDA 线上。

若此时数据寄存器和移位寄存器为空，则 TxE＝1，产生事件 EV8_1，写入数据到数据

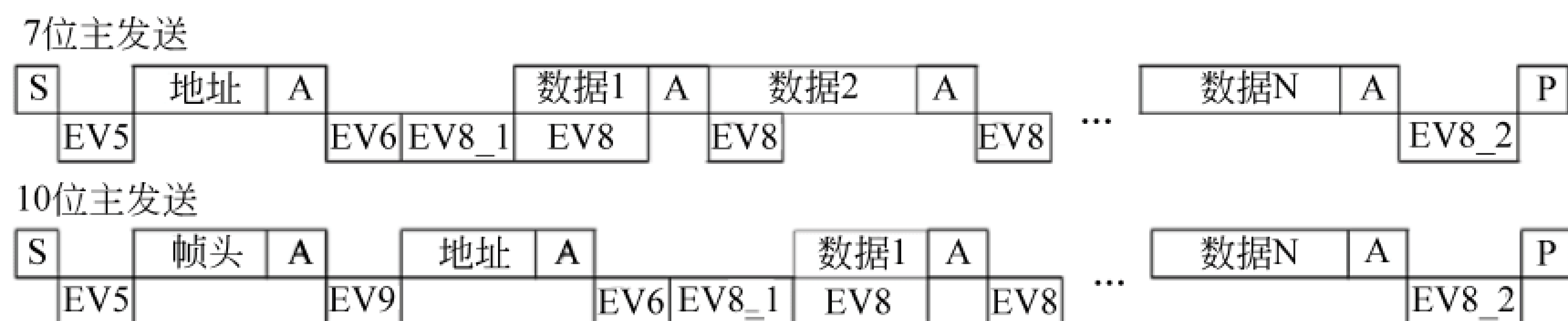


图 9-25　主模式下的发送时序

寄存器 DR 后，TxE＝0，此时由于移位寄存器为空，则数据立即被送到移位寄存器发送，TxE＝1，此时产生事件 EV8。当收到应答脉冲时，TxE 位被硬件置位，产生 EV8，如果 TxE 被置位并且在上一次数据发送结束之前没有写新的数据字节到 DR 寄存器，则 BTF 被硬件置位，产生事件 EV8_2，在清除 BTF 之前 I2C 接口将保持 SCL 为低电平；读出 I2C_SR1 之后再写入 I2C_DR 寄存器将清除 BTF 位。

在 DR 寄存器中写入最后一个字节后，通过设置 STOP 位产生一个停止条件，然后 I2C 接口将自动回到从模式(M/S 位清除)。

2. 主模式下的接收

主接收模式下，主设备向从设备发送起始信号和地址，然后接收从设备发来的数据。如图 9-26 所示，主设备首先发起起始信号，等待事件 EV5，然后发送地址和读命令，从设备返回 ACK 后产生 EV6 事件，清除 ADDR 之后，I2C 接口进入主接收器模式。在此模式下，I2C 接口从 SDA 线接收数据字节，并通过内部移位寄存器送至 DR 寄存器。在每个字节后，如果 ACK 位被置 1，发出一个应答脉冲，硬件设置 RxNE＝1，产生事件 EV7 如果设置了 INEVFEN 和 ITBUFEN 位，则会产生一个中断。如果 RxNE 位被置位，并且在接收新数据结束前，DR 寄存器中的数据没有被读走，硬件将设置 BTF＝1，在清除 BTF 之前 I2C 接口将保持 SCL 为低电平；读出 I2C_SR1 之后再读出 I2C_DR 寄存器将清除 BTF 位。

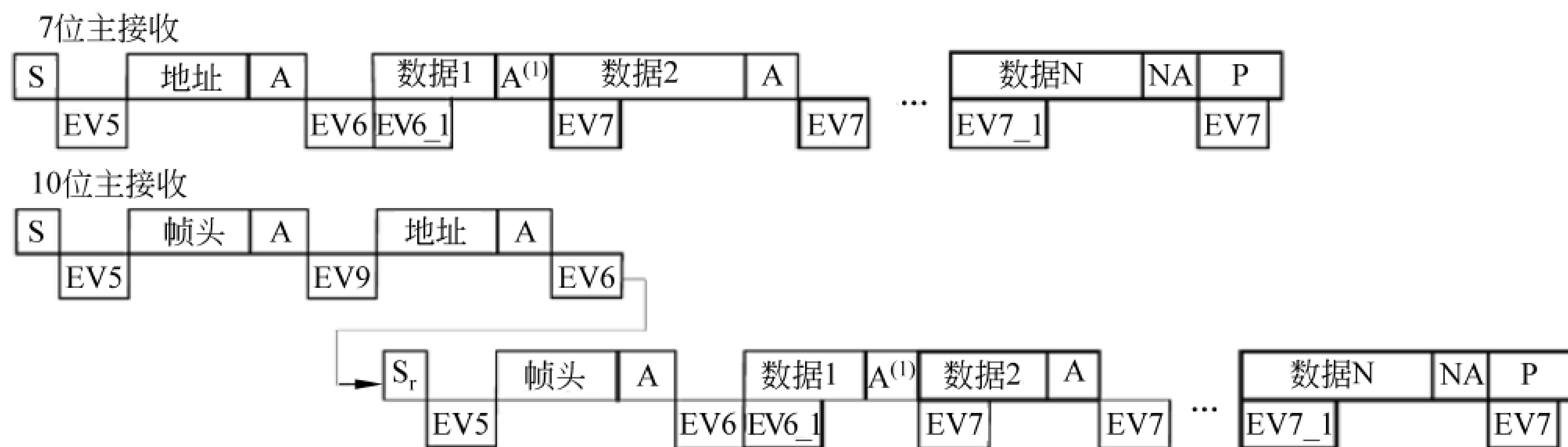


图 9-26　主接收模式时序

主设备在从从设备接收到最后一个字节后发送一个 NACK。接收到 NACK 后，从设备释放对 SCL 和 SDA 线的控制，主设备就可以发送下一个停止或重起始信号。为了在收到最后一个字节后产生一个 NACK 脉冲，在读倒数第二个数据字节之后(在倒数第二个 RxNE 事件之后)必须清除 ACK 位，并设置 STOP 位或者设置 START 位。如果只接收一个字节时，在第一个 EV6 事件要关闭应答和设置停止条件的产生位。在产生了停止条件

后，I2C接口自动回到从模式（M/SL位被清除）。

9.5.3 总线通信错误

I2C总线的通信可能会由于一些原因造成通信失败。

1. 总线错误（BERR）

在一个地址或数据字节传输期间，当I2C接口检测到一个外部的停止或起始条件则产生总线错误，BERR位被置位为1，如果设置了ITERREN位，则产生一个中断。

在主模式情况下，硬件不释放总线，同时不影响当前的传输状态。此时由软件决定是否要中止当前的传输。在从模式情况下，数据被丢弃，硬件释放总线：

(1) 如果是错误的开始条件，从设备认为是一个重启动，并等待地址或停止条件。

(2) 如果是错误的停止条件，从设备按正常的停止条件操作，同时硬件释放总线。

2. 应答错误（AF）

当接口检测到一个无应答位时，产生应答错误，AF位被置位，如果设置了ITERREN位，则产生一个中断。主发送模式收到NACK时，需要产生一个停止条件，从发送模式收到NACK时，释放总线。

3. 仲裁丢失（ARLO）

多主设备通信情况下，当I2C接口检测到仲裁丢失时产生仲裁丢失错误，ARLO位被硬件置位，如果设置了ITERREN位，则产生一个中断，I2C接口自动回到从模式（M/SL位被清除），释放总线。

4. 过载/欠载错误（OVR）

在从模式下，如果禁止时钟延长（NOSTRETCH＝1），I2C接口正在接收数据时，当它已经接收到一个字节（RxNE＝1），但在DR寄存器中前一个字节数据还没有被读出，则发生过载错误。此时，最后接收的数据被丢弃；I2C接口正在发送数据时，在下一个字节的时钟到达之前，新的数据还未写入DR寄存器（TxE＝1），则发生欠载错误。此时，DR寄存器中的前一个字节将被重复发出。过载和欠载时，用户需要自己控制发送端是否重发因接收端过载丢失的数据，还是接收端丢失因发送端欠载重复发送的数据。

如果允许时钟延长（NOSTRETCH＝0），发送器模式下，如果TxE＝1且BTF＝1，I2C接口在传输前保持时钟线为低，以等待软件读取SR1，然后把数据写进数据寄存器（缓冲器和移位寄存器都是空的）；接收器模式：如果RxNE＝1且BTF＝1，I2C接口在接收到数据字节后保持时钟线为低，以等待软件读SR1，然后读数据寄存器DR（缓冲器和移位寄存器都是满的）。因此允许时钟延长实际上是一种流量控制的方法，可以解决过载和欠载问题。

9.5.4 中断请求

I2C的中断源如表9-2所示。

表 9-2　I2C 中断请求表

中 断 事 件	事 件 标 志	开启控制位
起始位已发送(主)	SB	ITEVFEN
地址已发送(主)或地址匹配(从)	ADDR	
10 位头段已发送(主)	ADD10	
已收到停止(从)	STOPF	
数据字节传输完成	BTF	
接收缓冲区非空	RxNE	ITEVFEN 和 ITBUFEN
发送缓冲区空	TxE	
总线错误	BERR	ITERREN
仲裁丢失(主)	ARLO	
响应失败	AF	
过载/欠载	OVR	
PEC 错误	PECERR	
超时/Tlow 错误	TIMEOUT	
SMBus 提醒	SMBALERT	

三个中断开关 ITEVFEN、ITBUFEN、ITERREN 的关系如图 9-27 所示，如果要使用 TxE 和 RxNE，则需要打开 ITEVFEN 和 ITBUFEN。每个 I2C 控制器连接到 NVIC 的中断线有两个，分别是 IT_EVENT 和 IT_ERROR。

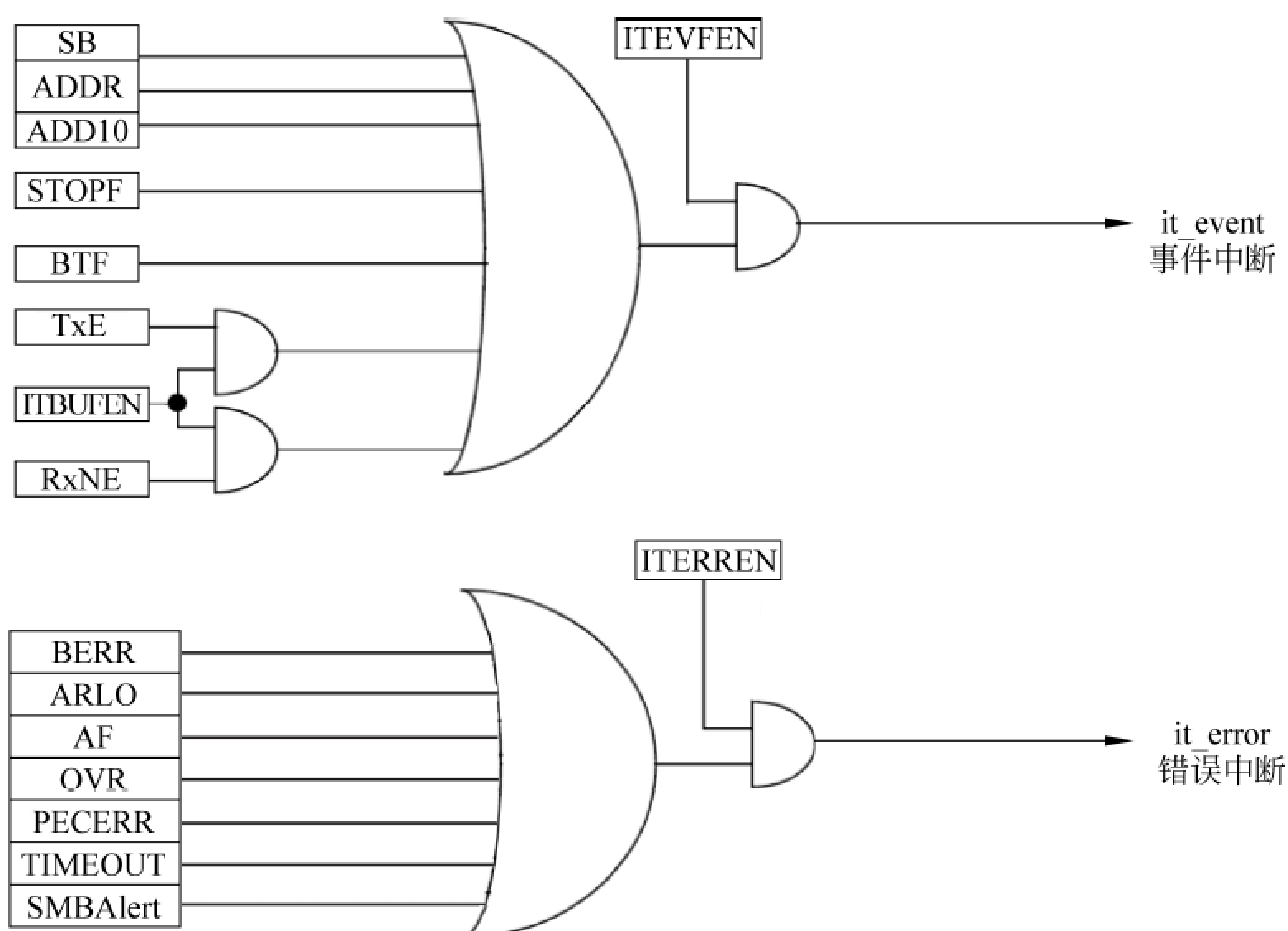


图 9-27　I2C 中断映射

I2C 用到的 I/O 引脚如表 9-3 所示。

表 9-3 IIC 外部引脚

I2C 引脚功能	I2C1 I/O 引脚	I2C2 I/O 引脚
I2C_SCL	PB6、PB8	PB10
I2C_SDA	PB7、PB9	PB11
I2C_SMBA	PB5	PB12

9.6 函数库

9.6.1 I2C 寄存器结构

I2C 寄存器结构，I2C_TypeDeff 在文件 stm32lxxx.h 中定义如下：

```
typedef struct
{
    __IO uint16_t CR1;                    // I2C 控制寄存器 1
    uint16_t RESERVED0;
    __IO uint16_t CR2;                    // I2C 控制寄存器 2
    uint16_t RESERVED1;
    __IO uint16_t OAR1;                   // I2C 自身地址寄存器 1
    uint16_t RESERVED2;
    __IO uint16_t OAR2;                   // I2C 自身地址寄存器 2
    uint16_t RESERVED3;
    __IO uint16_t DR;                     // I2C 数据寄存器
    uint16_t RESERVED4;
    __IO uint16_t SR1;                    // I2C 状态寄存器 1
    uint16_t RESERVED5;
    __IO uint16_t SR2;                    // I2C 状态寄存器 2
    uint16_t RESERVED6;
    __IO uint16_t CCR;                    // I2C 时钟控制寄存器
    uint16_t RESERVED7;
    __IO uint16_t TRISE;                  // I2C 上升时间寄存器
    uint16_t RESERVED8;
} I2C_TypeDef;
```

2 个 I2C 外设声明于文件 stm32l1xx.h 中：

```
#define PERIPH_BASE ((uint32_t)0x40000000)
#define APB1PERIPH_BASE PERIPH_BASE
```

```
#define APB2PERIPH_BASE (PERIPH_BASE + 0x10000)
#define AHBPERIPH_BASE (PERIPH_BASE + 0x20000)
#define I2C1_BASE (APB1PERIPH_BASE + 0x5400)
#define I2C2_BASE (APB1PERIPH_BASE + 0x5800)
#define I2C1 (I2C_TypeDef * ) I2C1_BASE
#define I2C2 (I2C_TypeDef * ) I2C2_BASE
```

用于 I2C 寄存器初始化的 I2C_InitTypeDef 结构体定义于文件 stm32l1xx_i2c.h：

```
typedef struct
{
  Uint32_t I2C_ClockSpeed;
  uint16_t I2C_Mode;
  uint16_t I2C_DutyCycle;
  uint16_t I2C_OwnAddress1;
  uint16_t I2C_Ack;
  uint16_t I2C_AcknowledgedAddress;
} I2C_InitTypeDef;
```

其中 I2C_ClockSpeed 参数用来设置时钟频率，这个值不能高于 400kHz。

I2C_Mode 用于设置 I2C 的模式，其取值范围为：

- I2C_Mode_I2C　　设置 I2C 为 I2C 模式
- I2C_Mode_SMBusDevice　　设置 I2C 为 SMBus 设备模式
- I2C_Mode_SMBusHost　　设置 I2C 为 SMBus 主控模式

I2C_DutyCycle 用以设置 I2C 的占空比，该参数只有在 I2C 工作在快速模式(时钟工作频率高于 100kHz)下才有意义，其取值范围为：

- I2C_DutyCycle_16_9　　I2C 快速模式 Tlow/Thigh=16/9
- I2C_DutyCycle_2　　I2C 快速模式 Tlow/Thigh=2

I2C_OwnAddress1 该参数用来设置第一个设备自身地址，它可以是一个 7 位地址或者一个 10 位地址。

I2C_Ack 使能或者失能应答(ACK)，其取值为：

- I2C_Ack_Enable　　使能应答(ACK)
- I2C_Ack_Disable　　失能应答(ACK)

I2C_AcknowledgedAddres 定义了应答 7 位地址还是 10 位地址，其取值范围为：

- I2C_AcknowledgeAddress_7bit　　应答 7 位地址
- I2C_AcknowledgeAddress_10bit　　应答 10 位地址

9.6.2 I2C 库函数

ST 提供的 I2C 标准函数库如表 9-4 所示。

表 9-4　I2C 函数列表

函　数　名	描　　述
I2C_DeInit	将外设 I2Cx 寄存器重设为默认值
I2C_Init	根据 I2C_InitStruct 中指定的参数初始化外设 I2Cx 寄存器
I2C_StructInit	把 I2C_InitStruct 中的每一个参数按默认值填入
I2C_Cmd	使能或者失能 I2C 外设
I2C_GenerateSTART	产生 I2Cx 传输 START 条件
I2C_GenerateSTOP	产生 I2Cx 传输 STOP 条件
I2C_AcknowledgeConfig	使能或者失能指定 I2C 的应答功能
I2C_OwnAddress2Config	设置指定 I2C 的自身地址 2
I2C_DualAddressCmd	使能或者失能指定 I2C 的双地址模式
I2C_GeneralCallCmd	使能或者失能指定 I2C 的广播呼叫功能
I2C_SoftwareResetCmd	使能或者失能指定 I2C 的软件复位
I2C_SMBusAlertConfig	驱动指定 I2Cx 的 SMBusAlert 引脚电平为高或低
I2C_ARPCmd	使能或者失能指定 I2C 的 ARP
I2C_StretchClockCmd	使能或者失能指定 I2C 的时钟延展
I2C_FastModeDutyCycleConfig	选择指定 I2C 的快速模式占空比
I2C_Send7bitAddress	向指定的从 I2C 设备传送地址字
I2C_SendData	通过外设 I2Cx 发送一个数据
I2C_ReceiveData	返回通过 I2Cx 最近接收的数据
I2C_NACKPositionConfig	2 字节接收时 NACK 的发送位置
I2C_TransmitPEC	使能或者失能指定 I2C 的 PEC 传输
I2C_PECPositionConfig	选择指定 I2C 的 PEC 位置
I2C_CalculatePEC	使能或者失能指定 I2C 的传输字 PEC 值计算
I2C_GetPEC	返回指定 I2C 的 PEC 值
I2C_DMACmd	使能或者失能指定 I2C 的 DMA 请求
I2C_DMALastTransferCmd	使下一次 DMA 传输为最后一次传输
I2C_ReadRegister	读取指定的 I2C 寄存器并返回其值
I2C_ITConfig	使能或者失能指定的 I2C 中断
I2C_CheckEvent	检查最近一次 I2C 事件是否是输入的事件
I2C_GetLastEvent	返回最近一次 I2C 事件
I2C_GetFlagStatus	检查指定的 I2C 标志位设置与否

续表

函 数 名	描 述
I2C_ClearFlag	清除 I2Cx 的待处理标志位
I2C_GetITStatus	检查指定的 I2C 中断发生与否
I2C_ClearITPendingBit	清除 I2Cx 的中断待处理位

在调用 I2C 库函数前，需要打开 I2C 总线时钟，调用 RCC_APB1PeriphClockCmd()。

1）函数 I2C_DeInit

功能描述：将外设 I2Cx 寄存器重设为默认值。

函数原型：void I2C_DeInit(I2C_TypeDef * I2Cx)。

输入参数 I2Cx：用来选择 I2C 外设，x 可以是 1 或者 2。

示例：

```
I2C_DeInit(I2C2);
```

2）函数 I2C_Init

功能描述：根据 I2C_InitStruct 中指定的参数初始化外设 I2Cx 寄存器。

函数原型：void I2C_Init(I2C_TypeDef * I2Cx，I2C_InitTypeDef * I2C_InitStruct)。

输入参数 I2Cx：用来选择 I2C 外设，x 可以是 1 或者 2。

输入参数 I2C_InitStruct：指向结构 I2C_InitTypeDef 的指针。

示例：

```
I2C_InitTypeDef I2C_InitStructure;
I2C_InitStructure.I2C_Mode = I2C_Mode_SMBusHost;
I2C_InitStructure.I2C_DutyCycle = I2C_DutyCycle_2;
I2C_InitStructure.I2C_OwnAddress1 = 0x03A2;
I2C_InitStructure.I2C_Ack = I2C_Ack_Enable;
I2C_InitStructure.I2C_AcknowledgedAddress =
I2C_AcknowledgedAddress_10bit;
I2C_InitStructure.I2C_ClockSpeed = 200000;
I2C_Init(I2C1, &I2C_InitStructure);
```

3）函数 I2C_StructInit

功能描述：把 I2C_InitStruct 中的每一个参数按默认值填入。

函数原型：void I2C_StructInit(I2C_InitTypeDef * I2C_InitStruct)。

输入参数 I2C_InitStruct，指向结构 I2C_InitTypeDef 的指针，待初始化。

I2C_InitStruct 各个成员的默认值为：

- I2C_Mode　　I2C_Mode_I2C
- I2C_DutyCycle　　I2C_DutyCycle_2
- I2C_OwnAddress1　　0

- I2C_Ack　　I2C_Ack_Disable
- I2C_AcknowledgedAddres　　I2C_AcknowledgedAddress_7bit
- I2C_ ClockSpeed　　5000

示例：

```
I2C_InitTypeDef I2C_InitStructure;
I2C_StructInit(&I2C_InitStructure);
```

4）函数 I2C_Cmd

功能描述：使能或者禁用 I2C 外设。

函数原型：void I2C_Cmd(I2C_TypeDef * I2Cx，FunctionalState NewState)。

输入参数 I2Cx：用来选择 I2C 外设，x 可以是 1 或者 2。

输入参数 NewState：外设 I2Cx 的状态，取值为 ENABLE 或者 DISABLE。

示例：

```
I2C_Cmd(I2C1,ENABLE);
```

5）函数 I2C_GenerateSTART

功能描述：产生 I2Cx 的 START 信号。

函数原型：void I2C_GenerateSTART(I2C_TypeDef * I2Cx，FunctionalState NewState)。

输入参数 I2Cx：用来选择 I2C 外设，x 可以是 1 或者 2。

输入参数 NewState，I2Cx START 条件的状态，取值为 ENABLE 或者 DISABLE。

示例：

```
I2C_GenerateSTART(I2C1,ENABLE);
```

6）函数 I2C_GenerateSTOP

功能描述：产生 I2Cx 的 STOP 信号。

函数原型：void I2C_GenerateSTOP(I2C_TypeDef * I2Cx，FunctionalState NewState)。

输入参数 I2Cx：用来选择 I2C 外设，x 可以是 1 或者 2。

输入参数 NewState：表示 I2Cx STOP 条件的新状态，取值为 ENABLE 或者 DISABLE。

示例：

```
I2C_GenerateSTOP(I2C2,ENABLE);
```

7）函数 I2C_AcknowledgeConfig

功能描述：使能或者失能指定 I2C 控制器的应答功能。

函数原型：void I2C_AcknowledgeConfig(I2C_TypeDef * I2Cx，FunctionalState NewState)。

输入参数 I2Cx：用来选择 I2C 外设，x 可以是 1 或者 2。

输入参数 NewState：表示 I2Cx STOP 条件的状态，取值为 ENABLE 或者 DISABLE。

示例：

```
I2C_AcknowledgeConfig(I2C1,ENABLE);
```

8) I2C_OwnAddress2Config 函数

功能描述：配置 I2C 自身地址 2。

函数原型：void I2C_OwnAddress2Config(I2C_TypeDef * I2Cx, uint8_t Address)。

输入参数 I2Cx：用来选择 I2C 外设，x 可以是 1 或者 2。

输入参数 Address：I2C 的 7 位地址自身地址 2 的值。

示例：

```
I2C_OwnAddress2Config(I2C1,0x31);
```

9) I2C_DualAddressCmd 函数

功能描述：使能或禁用 I2C 的双地址模式。

函数原型：void I2C_DualAddressCmd(I2C_TypeDef * I2Cx, FunctionalState NewState)。

输入参数 I2Cx：用来选择 I2C 外设，x 可以是 1 或者 2。

输入参数 NewState，表示 I2Cx 双地址模式的状态，取值为 ENABLE 或者 DISABLE。

示例：

```
I2C_DualAddressCmd(I2C1,ENABLE);
```

10) 函数 I2C_SoftwareResetCmd

功能描述：使能或者禁用指定 I2C 的软件复位。

函数原型：I2C_SoftwareResetCmd(I2C_TypeDef * I2Cx, FunctionalState NewState)。

输入参数 I2Cx：用来选择 I2C 外设，x 可以是 1 或者 2。

输入参数 NewState 为软件复位的新状态，取值为 ENABLE 或者 DISABLE。

示例：

```
I2C_SoftwareResetCmd(I2C1,ENABLE);
```

11) 函数 I2C_StretchClockCmd

功能描述：使能或者禁用指定 I2C 的时钟延长功能。

函数原型：void I2C_StretchClockCmd(I2C_TypeDef * I2Cx,FunctionalState NewState)。

输入参数 I2Cx：用来选择 I2C 外设，x 可以是 1 或者 2。

输入参数 NewState 为时钟延长功能的状态，取值为 ENABLE 或者 DISABLE。

示例：

```
I2C_StretchClockCmd(I2C1,ENABLE);
```

12) 函数 I2C_FastModeDutyCycleConfig

功能描述：选择指定 I2C 的快速模式占空比。

函数原型：void I2C_FastModeDutyCycleConfig(I2C_TypeDef * I2Cx, uint16_t I2C_DutyCycle)。

输入参数 I2Cx：用来选择 I2C 外设，x 可以是 1 或者 2。

输入参数 I2C_DutyCycle 用来指定快速模式占空比，其取值为：

- I2C_DutyCycle_16_9　I2C 快速模式 Tlow/Thigh = 16/9
- I2C_DutyCycle_2　I2C 快速模式 Tlow/Thigh = 2

示例：

```
I2C_FastModeDutyCycleConfig(I2C2, I2C_DutyCycle_16_9);
```

13）函数 I2C_Send7bitAddress

功能描述：向指定的从 I2C 设备传送地址字。

函数原型：void I2C_Send7bitAddress(I2C_TypeDef * I2Cx, uint8_t Address, uint8_t I2C_Direction)。

输入参数 I2Cx：用来选择 I2C 外设，x 可以是 1 或者 2。

输入参数 Address 为待传输的从 I2C 设备地址。

输入参数 I2C_Direction 用于设置指定的 I2C 设备工作为发送模式还是接收模式，其取值为：

- I2C_Direction_Transmitter　选择发送模式
- I2C_Direction_Receiver　选择接收模式

示例：

```
I2C_Send7bitAddress(I2C1, 0xA8, I2C_Direction_Transmitter);
```

14）函数 I2C_SendData

功能描述：通过外设 I2Cx 发送一个数据。

函数原型：void I2C_SendData(I2C_TypeDef * I2Cx, uint8_t Data)。

输入参数 I2Cx：用来选择 I2C 外设，x 可以是 1 或者 2。

输入参数 Data 为待发送的数据。

示例：

```
I2C_SendData(I2C2,0x5D);
```

15）函数 I2C_ ReceiveData

功能描述：返回通过 I2Cx 最近接收的数据。

函数原型：uint8_t I2C_ReceiveData(I2C_TypeDef * I2Cx)。

输入参数 I2Cx：用来选择 I2C 外设，x 可以是 1 或者 2。

示例：

```
uint8_t ReceivedData;
ReceivedData = I2C_ReceiveData(I2C1);
```

16）函数 I2C_NACKPositionConfig

功能描述：在主模式接收时，两个字节读取下确认 ACK 的发送时机。

函数原型：uint8_t I2C_ReceiveData(I2C_TypeDef * I2Cx)。

输入参数 I2Cx：用来选择 I2C 外设，x 可以是 1 或者 2。

输入参数 I2C_NACKPosition，指定 NACK 的发送位置，其取值为：

- I2C_NACKPosition_Next　　下一个字节接收后发 NACK
- I2C_NACKPosition_Current　　当前字节接收后发 NACK

示例：

```
I2C_NACKPositionConfig(I2C1,I2C_NACKPosition_Next);
```

17）函数 I2C_ ReadRegister

功能描述：读取指定的 I2C 寄存器并返回其值。

函数原型：uint16_t I2C_ReadRegister(I2C_TypeDef * I2Cx，uint8_t I2C_Register)。

输入参数 I2Cx：用来选择 I2C 外设，x 可以是 1 或者 2。

输入参数 I2C_Register 指定待读取的 I2C 寄存器，其取值为：I2C_Register_CR1、I2C_Register_CR2、I2C_Register_OAR1、I2C_Register_OAR2、I2C_Register_DR、I2C_Register_SR1、I2C_Register_SR2、I2C_Register_CCR、I2C_Register_TRISE。

返回值为被读取的寄存器值。

示例：

```
uint16_t RegisterValue;
RegisterValue = I2C_ReadRegister(I2C2, I2C_Register_CR1);
```

18）函数 I2C_ITConfig

功能描述：使能或者禁用指定的 I2C 中断。

函数原型：void I2C_ITConfig(I2C_TypeDef * I2Cx，uint16_t I2C_IT，FunctionalState NewState)。

输入参数 I2Cx：用来选择 I2C 外设，x 可以是 1 或者 2。

输入参数 I2C_IT 为待使能或者禁用的 I2C 中断源，可以取一个或者多个取值的组合作为该参数的值。

- I2C_IT_BUF　缓存中断屏蔽。
- I2C_IT_EVT　事件中断屏蔽。
- I2C_IT_ERR　错误中断屏蔽。

输入参数 NewState 表示 I2Cx 中断的新状态，取值为 ENABLE 或 DISABLE。

示例：

```
I2C_ITConfig(I2C2, I2C_IT_BUF | I2C_IT_EVT,ENABLE);
```

19）函数 I2C_GetLastEvent

功能描述：返回最近一次 I2C 事件。

函数原型：uint32_t I2C_GetLastEvent(I2C_TypeDef * I2Cx)。

输入参数 I2Cx：用来选择 I2C 外设，x 可以是 1 或者 2。

返回值为最近一次 I2C 事件,事件定义见表 9-5。

示例:

```
uint32_t Event;
Event = I2C_GetLastEvent(I2C1);
```

表 9-5　I2C 事件定义

I2C_Event	描　　述
I2C_EVENT_SLA VE_RECEIVER_ADDRESS_MA TCHED	EV1
I2C_EVENT_SLAVE_TRANSMITTER_ADDRESS_MATCHED	EV1
I2C_EVENT_SLAVE_RECEIVER_SECONDADDRESS_MATCHED	EV1
I2C_EVENT_SLAVE_TRANSMITTER_SECONDADDRESS_MA TCHED	EV1
I2C_EVENT_SLAVE_GENERALCALLADDRESS_MATCHED	EV1
I2C_EVENT_SLA VE_BYTE_RECEIVED	EV2
I2C_EVENT_SLAVE_BYTE_TRANSMITTED	EV3
I2C_EVENT_SLAVE_ACK_FAILURE	EV3～1
I2C_EVENT_SLA VE_STOP_DETECTED	EV4
I2C_EVENT_MASTER_MODE_SELECT	EV5
I2C_EVENT_MASTER_RECEIVER_MODE_SELECTED	EV6
I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED	EV6
I2C_EVENT_MASTER_BYTE_RECEIVED	EV7
I2C_EVENT_MASTER_BYTE_TRANSMITTED	EV8
I2C_EVENT_MASTER_MODE_ADDRESS10	EV9

20) 函数 I2C_CheckEvent

功能描述:检查最近一次 I2C 事件是否是输入的事件。

函数原型:ErrorStatus I2C_CheckEvent(I2C_TypeDef * I2Cx, uint32_t I2C_EVENT)。

输入参数 I2Cx:用来选择 I2C 外设,x 可以是 1 或者 2。

输入参数 I2C_Event 用于指定待检查的事件,其取值见表 9-5。

返回值 ErrorStatus 取值为 SUCCESS 或 ERROR,如果是 SUCCESS 表明所检查的事件是最近一次 I2C 事件,ERROR 表明最近一次 I2C 事件不是所检查的事件。

示例:

```
ErrorStatus Status;
Status = I2C_CheckEvent(I2C1,I2C_EVENT_MSTER_BYTE_RECEIVED);
```

21）函数 I2C_GetFlagStatus

功能描述：检查指定的 I2C 标志位设置与否。

函数原型：FlagStatus I2C_GetFlagStatus(I2C_TypeDef * I2Cx，uint32_t I2C_FLAG)。

输入参数 I2Cx 用来选择 I2C 外设，I2C_FLAG 用于指定待检查的 I2C 标志位，其取值范围如表 9-6 所示。

返回值为 I2C_FLAG 的状态，取值范围为 SET 或 RESET。

表 9-6 I2C_FLAG 的取值

I2C_FLAG	描　述
I2C_FLAG_DUALF	双标志位(从模式)
I2C_FLAG_SMBHOST	SMBus 主报头(从模式)
I2C_FLAG_SMBDEFAULT	SMBus 缺省报头(从模式)
I2C_FLAG_GENCALL	广播报头标志位(从模式)
I2C_FLAG_TRA	发送/接收标志位
I2C_FLAG_BUSY	总线忙标志位
I2C_FLAG_MSL	主/从标志位
I2C_FLAG_SMBALERT	SMBus 报警标志位
I2C_FLAG_TIMEOUT	超时或者 Tlow 错误标志位
I2C_FLAG_PECERR	接收 PEC 错误标志位
I2C_FLAG_OVR	溢出/不足标志位(从模式)
I2C_FLAG_AF	应答错误标志位
I2C_FLAG_ARLO	仲裁丢失标志位(主模式)
I2C_FLAG_BERR	总线错误标志位
I2C_FLAG_TXE	数据寄存器空标志位(发送端)
I2C_FLAG_RXNE	数据寄存器非空标志位(接收端)
I2C_FLAG_STOPF	停止标志位(从模式)
I2C_FLAG_ADD10	10 位报头发送(主模式)
I2C_FLAG_BTF	字传输完成标志位
I2C_FLAG_ADDR	地址发送标志位(主模式)或地址匹配标志位(从模式)ADDR
I2C_FLAG_SB	起始位标志位(主模式)

示例：

```
Flagstatus Status;
Status = I2C_GetFlagStatus(I2C2,I2C_FLAG_AF);
```

22）函数 I2C_ClearFlag

功能描述：清除 I2Cx 的待处理标志位。

函数原型：void I2C_ClearFlag(I2C_TypeDef * I2Cx, uint32_t I2C_FLAG)。

输入参数 I2Cx 用来选择 I2C 外设，I2C_FLAG 为待清除的 I2C 标志位，取值见表 9-6。但 DUALF，SMBHOST，SMBDEFAULT，GENCALL，TRA，BUSY，MSL，TXE 和 RXNE 不能被本函数清除。

示例：

```
I2C_ClearFlag(I2C2, I2C_FLAG_STOPF);
```

23）函数 I2C_GetITStatus

功能描述：检查指定的 I2C 中断发生与否。

函数原型：ITStatus I2C_GetITStatus(I2C_TypeDef * I2Cx, uint32_t I2C_IT)。

输入参数 I2Cx 用来选择 I2C 外设，I2C_IT 为待检查的 I2C 中断源，其取值见表 9-7。

返回值为 I2C_IT 的状态，取值为 SET 或者 RESET。

表 9-7　I2C_IT 值

I2C_IT	描　　述
I2C_IT_SMBALERT	SMBus 报警标志位
I2C_IT_TIMEOUT	超时或者 Tlow 错误标志位
I2C_IT_PECERR	接收 PEC 错误标志位
I2C_IT_OVR	溢出/不足标志位(从模式)
I2C_IT_AF	应答错误标志位
I2C_IT_ARLO	仲裁丢失标志位(主模式)
I2C_IT_BERR	总线错误标志位
I2C_IT_STOPF	停止探测标志位(从模式)
I2C_IT_ADD10	10 位报头发送(主模式)
I2C_IT_BTF	字传输完成标志位
I2C_IT_ADDR	地址发送标志位(主模式)与地址匹配标志位(从模式)ADDR
I2C_IT_SB	起始位标志位(主模式)

示例：

```
ITstatus Status;
Status = I2C_GetITStatus(I2C1, I2C_IT_OVR);
```

24）函数 I2C_ClearITPendingBit

功能描述：清除 I2Cx 的中断待处理位

函数原型：void I2C_ClearITPendingBit(I2C_TypeDef * I2Cx, uint32_t I2C_IT)。

输入参数 I2Cx 用来选择 I2C 外设,I2C_IT 指定待清除的 I2C 中断源,取值见表 9-6。

示例:

```
I2C_ClearITPendingBit(I2C2, I2C_IT_TIMEOUT);
```

9.7 I2C 案例

9.7.1 I2C 寄存器操作案例

1) I2C_Init 函数的实现

```
void I2C_Init(I2C_TypeDef * I2Cx, I2C_InitTypeDef * I2C_InitStruct)
{
  uint16_t tmpreg = 0, freqrange = 0;
  uint16_t result = 0x04;
  uint32_t pclk1 = 8000000;
  RCC_ClocksTypeDef  rcc_clocks;
  // CR2 寄存器配置
  tmpreg = I2Cx->CR2;                    //获取 CR2 的值
  // 清除频率字段 FREQ[5:0]
  tmpreg &= (uint16_t)~ ((uint16_t)I2C_CR2_FREQ);
  // 配置频率字段,首选获取 APB1 总线时钟
  RCC_GetClocksFreq(&rcc_clocks);
  pclk1 = rcc_clocks.PCLK1_Frequency;
  //将时钟值除以 1M,得到一个整数,作为 FREQ[5:0]的配置,实际频率即为总线频率
  freqrange = (uint16_t) (pclk1 / 1000000);
  tmpreg |= freqrange;
  //写会到 CR2 寄存器中,频率配置完成
  I2Cx->CR2 = tmpreg;
  // CCR 寄存器配置,配置前首先禁用 I2C
  I2Cx->CR1 &= (uint16_t)~ ((uint16_t)I2C_CR1_PE);
  //将 tmpreg 清 0,即 F/S, DUTY 和 CCR[11:0] 均置为 0
  tmpreg = 0;
  //标准模式下的配置
  if (I2C_InitStruct->I2C_ClockSpeed <= 100000)
  {
    //通过时钟频率计算 CCR 的值,最小值为 4
    result = (uint16_t) (pclk1 / (I2C_InitStruct->I2C_ClockSpeed << 1));
    if (result < 0x04)
      result = 0x04;
    tmpreg |= result;        //写入 CCR
```

```
    //配置上升沿时间为 I2C的 FREQ字段值+1,即最大上升时间为:1000ns
    I2Cx->TRISE = freqrange + 1;
  }
  //如果是快速模式, PCLK1必须是 10 MHz的整数倍 (FREQ字段配置)
  else                    //(I2C_InitStruct->I2C_ClockSpeed <= 400000)
  { //快速模式下需要配置占空比,不同占空比计算 CCR的值不一样
    if (I2C_InitStruct->I2C_DutyCycle == I2C_DutyCycle_2)
    {
      result = (uint16_t) (pclk1 / (I2C_InitStruct->I2C_ClockSpeed * 3));
    }
    else                  //I2C_InitStruct->I2C_DutyCycle == I2C_DutyCycle_16_9
    {
        result = (uint16_t) (pclk1 / (I2C_InitStruct->I2C_ClockSpeed * 25));
        result |= I2C_DutyCycle_16_9;
    }
    //CCR最小值不能小于 1
    if ((result & I2C_CCR_CCR) == 0)
      result |= (uint16_t)0x0001;
    //设置快速模式位和最大上升时延
    tmpreg |= (uint16_t) (result | I2C_CCR_FS);
    I2Cx->TRISE = (uint16_t) (((freqrange * (uint16_t)300) / (uint16_t)1000) + (uint16_t)1);
  }
  //写入 CCR,使能 I2C
  I2Cx->CCR = tmpreg;
  I2Cx->CR1 |= I2C_CR1_PE;
  //CR1寄存器配置
  tmpreg = I2Cx->CR1;
  //清除 ACK, SMBTYPE 和 SMBUS 位
  tmpreg &= CR1_CLEAR_MASK;
  //根据 I2C_Mode value 和 I2C_Ack value 配置 CR1 寄存器
  tmpreg |= (uint16_t) ((uint32_t)I2C_InitStruct->I2C_Mode
            | I2C_InitStruct->I2C_Ack);
  I2Cx->CR1 = tmpreg;
  // OAR1 寄存器配置
  I2Cx->OAR1 = (I2C_InitStruct->I2C_AcknowledgedAddress
                | I2C_InitStruct->I2C_OwnAddress1);
}
```

9.7.2 I2C 基本配置

1) 基本配置流程

- 使能时钟 RCC_APB1PeriphClockCmd(RCC_APB1Periph_I2Cx，ENABLE)；
- 使能 SDA，SCL 所使用的 GPIO 端口时钟 RCC_AHBPeriphClockCmd()；

- 使用 GPIO_PinAFConfig()将 GPIO 和 I2C 复用功能进行映射；
- 配置 GPIO：复选功能，输出模式，类型为开漏，调用 GPIO_Init()初始化；
- 通过配置模式，占空比、地址、ACK 等参数，调用 I2C_Init()初始化；
- 如有必要可以单独调用 I2C_AcknowledgeConfig()、I2C_DualAddressCmd()、I2C_FastModeDutyCycleConfig()等函数进行配置；
- 如需中断，配置 NVIC 相应的中断源，并调用 I2C_ITConfig()开启所需的中断；
- 如需 DMA，需要调用 DMA_Init()、I2C_DMACmd()或 I2C_DMALastTransferCmd()；
- 调用 I2C_Cmd()使能 I2C。

2）基本配置案例

使用库函数实现 I2C 的配置和读写时序的三个函数如下：

```
void RCC_Configuration(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    I2C_InitTypeDef I2C_InitStructure;
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOB,ENABLE);
    RCC_AHB1PeriphClock(RCC_APB1Periph_I2C1,ENABLE);
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_7 | GPIO_Pin_6;
    GPIO_InitStructure.GPIO_Speed= GPIO_Speed_40MHz;
    GPIO_InitStructure.GPIO_Mode= GPIO_Mode_AF;
    GPIO_InitStructure.GPIO_OType= GPIO_OType_OD;
    GPIO_Init(GPIOB,&GPIO_InitStructure);
    GPIO_PinAFConfig(GPIOB,GPIO_PinSource7,GPIO_AF_I2C1);
    GPIO_PinAFConfig(GPIOB,GPIO_PinSource6,GPIO_AF_I2C1);
    I2C_InitStructure.I2C_Mode= I2C_Mode_I2C;
    I2C_InitStructure.I2C_DutyCycle= I2C_DutyCycle_2;
    I2C_InitStructure.OwnAddress1 = 0xA3;
    I2C_InitStructure.I2C_Ack= I2C_Ack_Enable;
    I2C_InitStructure.I2C_AcknowledgedAddress= I2C_AcknowledgedAddress_7bit;
    I2C_InitStructure.I2C_ClockSpeed = 200000;
    I2C_Init(I2C1,&I2C_InitStructure);
    I2C_Cmd(I2C1,ENABLE);
}
unsigned char I2C_ReadByte(unsigned char Address)
{   //等待 I2C 不忙
    while(I2C_GetFlagStatus(I2C1, I2C_FLAG_BUSY));
    I2C_GenerateSTART(I2C1, ENABLE);                                        //重新发送
    while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_MODE_SELECT));             //EV5
    I2C_Send7bitAddress(I2C1, Address, I2C_Direction_Receiver);             //发送地址
    //EV6
    while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_RECEIVER_MODE_SELECTED));
    //关闭应答和停止条件产生
```

```
        I2C_AcknowledgeConfig(I2C1, DISABLE);
        I2C_GenerateSTOP(I2C1, ENABLE);
        //等待 EV7
        while(!(I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_RECEIVED)));
        return = I2C_ReceiveData(I2C1);
}
void I2C_WriteByte(unsigned char buff, unsigned char Address)
{
        //产生起始条件
        I2C_GenerateSTART(I2C1,ENABLE);
        while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_MODE_SELECT));
        //向设备发送设备地址
        I2C_Send7bitAddress(I2C1,Address,I2C_Direction_Transmitter);
        //等待 ACK
    while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED));
    I2C_SendData(I2C1, buff);
    //发送完成
    while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_TRANSMITTED));
    //产生结束信号
     I2C_GenerateSTOP(I2C1, ENABLE);
}
```

9.7.3 模拟 I2C 实现

在很多情况下,我们经常使用 GPIO 来模拟 I2C 的时序,这样的方式使得程序的移植更为方便。在 I/O 模拟时,I2C 的总线无需一定要上拉电阻,可以将 I/O 配置为推挽模式。建议配置成开漏,配置上拉电阻,这样无需对端口进行输入输出切换。

```
#define I2C_SLAVE_ADDRESS7 0xA6
#define I2C_SCL_0 GPIO_ResetBits(GPIOB,GPIO_Pin_10)
#define I2C_SCL_1 GPIO_SetBits(GPIOB,GPIO_Pin_10)
#define I2C_SDA_0 GPIO_ResetBits(GPIOB,GPIO_Pin_11)
#define I2C_SDA_1 GPIO_SetBits(GPIOB,GPIO_Pin_11)
#define I2C_SDA_STAT GPIO_ReadInputDataBit(GPIOB,GPIO_Pin_11)
#define I2C_ACK 0
#define I2C_NACK 1
#define I2C_READY 0
#define I2C_BUSY 1
#define I2C_ERROR 3
void NOP(void)
{
        uint8_t i = 5;
```

```
    while(i--);
}
void TWI_Initialize(void)                       //没有上拉电阻将 SDA 和 SCL 设置成推挽输出
{
    GPIO_InitTypeDef GPIO_InitStructure;
    GPIO_InitStructure.GPIO_Speed=GPIO_Speed_10MHz;
    GPIO_InitStructure.GPIO_OType=GPIO_OType_PP;
    GPIO_InitStructure.GPIO_Pin=GPIO_Pin_10 | GPIO_Pin_11;
    GPIO_Init(GPIOB,&GPIO_InitStructure);
}
uint8_t I2C_Start(void)
{
    I2C_SDA_1;
    NOP();
    I2C_SCL_1;
    NOP();
    I2C_SDA_0;
    NOP();
    I2C_SCL_0;
    NOP();
    return I2C_READY;
}
void I2C_STOP(void)
{
    I2C_SDA_0;
    NOP();
    I2C_SCL_1;
    NOP();
    I2C_SDA_1;
    NOP();
}
uint8_t I2C_SendByte(uint8_t data)
{
    uint8_t i,err;
    I2C_SCL_0;
    for(i=0;i<8;i++)
    {
        if(data&0x80)
           I2C_SDA_1;
        else
           I2C_SDA_0;
        data<<=1;
        NOP();                                  //产生一个上升沿
```

```
        I2C_SCL_1;
        NOP();
        I2C_SCL_0;
        NOP();
    }
    I2C_SDA_1;                                      //接收从机应答
    NOP();
    I2C_SCL_1;
    NOP();
    while(I2C_SDA_STAT)
    {
        err++;
        if(err>250)
        {
            I2C_SCL_0;
            I2C_SDA_1;
            return I2C_NACK;
        }
    }
    I2C_SCL_0;
    I2C_SDA_1;
    return I2C_ACK;
}
uint8_t I2C_RecieveByte(void)
{
    uint8_t i,data;
    I2C_SDA_1;
    I2C_SCL_0;
    data=0;
    for(i=0;i<8;i++)
    {
        I2C_SCL_1;
        NOP();
        data<<=1;
        if(I2C_SDA_STAT)
            data|=0x01;
        I2C_SCL_0;
        NOP();
    }
    return data;
}
```

9.7.4 串行 Flash 通信

I2C 的基本时序能够实现两个设备之间的通信，但在实际应用中，我们通常用 MCU 作为主设备去和一些从设备进行数据交换，这些从设备的操作通常需要传输多个字节，例如图 9-28 所示的 SPI 接口串行 Flash AT24C02，存储大小为 256 字节共计 2kb，每个字节均可以随机进行读写访问，此时对于该设备的访问，我们需要两个地址，一个为 I2C 设备地址，用于在 I2C 总线上对 AT24C02 进行寻址，AT24C02 的设备地址的高四位为 1010，低三位由芯片外围的电路连接决定（A2、A1、A0 的电平）；另一个地址为 Flash 的存储地址，即要读写 AT24C02 的 256 个单元的某一个单元。

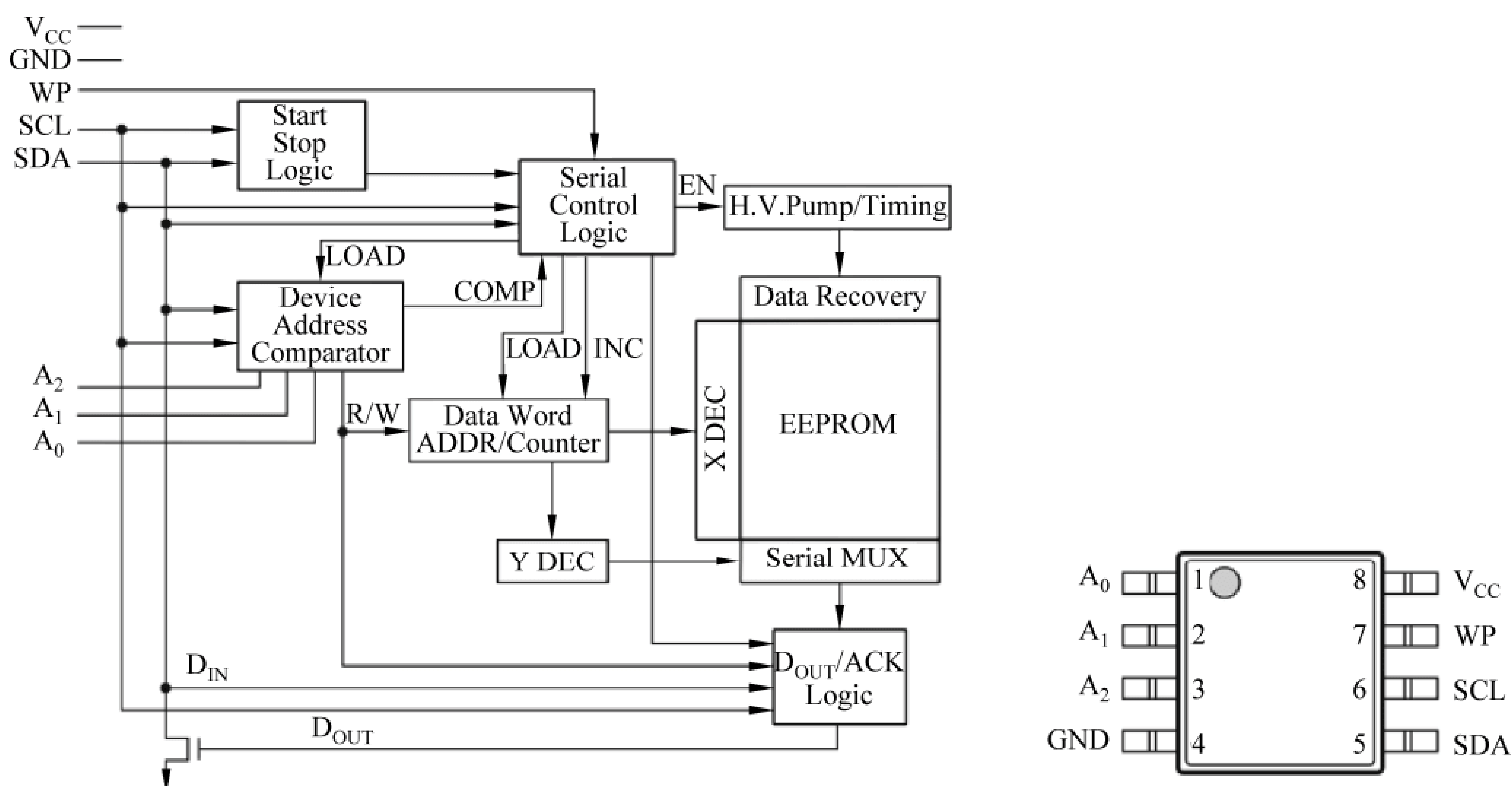


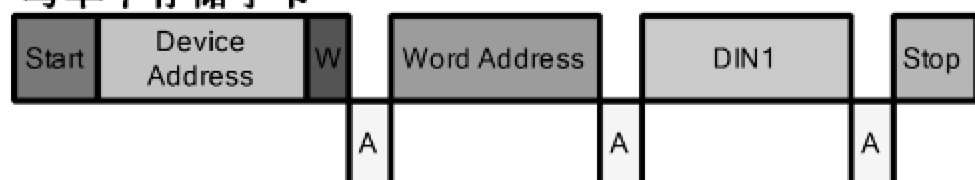
图 9-28 AT24C02 Flash 芯片

AT24C02 的操作时序见图 9-29。当需要往 Flash 写入数据时，我们首先发送 I2C 设备地址匹配 AT24C02，然后要发送 Flash 的写入地址，最后跟随一个或多个需要写入的数据，如图 9-29 的写单个存储字节和写多个存储字节所示。当需要从 Flash 读数据时，首先发送 I2C 地址匹配 AT24C02，此时我们需要先把要读的 Flash 地址写到 AT24C02，AT24C02 即可准备所要读的地址的数据，由于从设备无法主动发起数据，因此需要主设备再次发送 I2C 设备地址切换读写方向，发送一个读命令，从设备匹配地址后将准备好的数据发送到主设备，如图 9-29 所示的读单个字节和读多个字节时序。

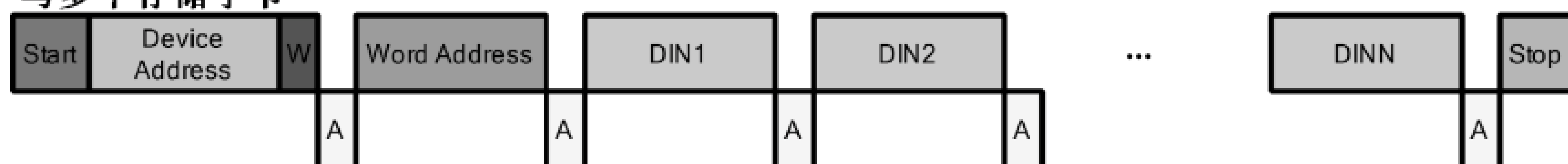
程序实现的伪代码如下：

```
uint8_t E2promWriteByte( uint16_t flash_addr, uint8_t data )
{
    I2CStart();
```

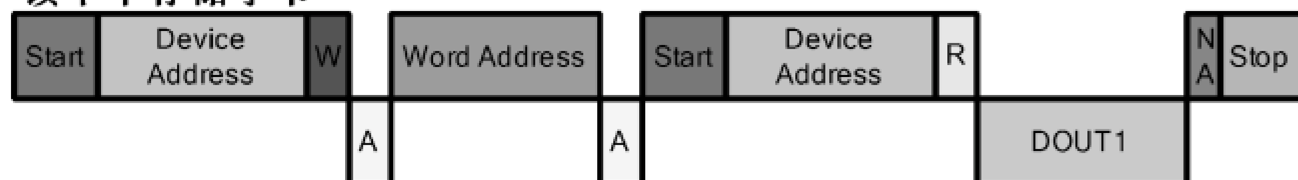
写单个存储字节



写多个存储字节



读单个存储字节



读多个存储字节

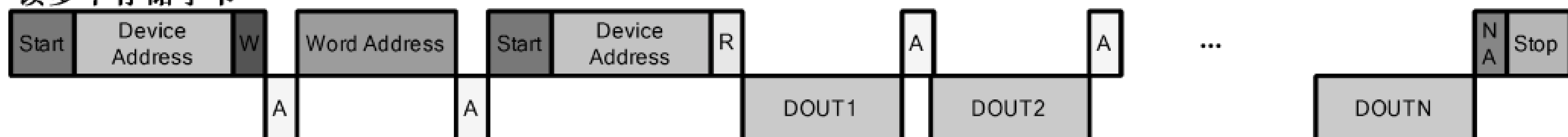




图 9-29　AT24C02 Flash 芯片读写时序

```
        I2CWriteByte( AT24C02ADDR_WRITE );
        WaitI2CSlaveAck();
        I2CWriteByte( flash_addr & 0xFF);
        WaitI2CSlaveAck();
        I2CWriteByte( data );
        WaitI2CSlaveAck();
        I2CStop();
        return 1;
    }
    uint8_t E2promReadByte( uint16_t flash_addr )
    {
        unsigned char ReadValue;
        I2CStart();
        I2CWriteByte(AT24C02ADDR_WRITE );
        WaitI2CSlaveAck();
        I2CWriteByte( flash_addr & 0xFF );
        WaitI2CSlaveAck();
        I2CStart();                                    //切换方向
        I2CWriteByte(AT24C02ADDR_READ );
```

```
    WaitI2CSlaveAck();
    ReadValue = I2CReadByte();                    //读数据
    I2CStop();
    return ReadValue;
}
```

9.7.5 ADT7420 温度传感器通信

ADT7420 是一款 I2C 接口的数字温度传感器，可以通过 I2C 直接读取到转换后的温度值，其电路连接如图 9-30 所示，SCL 和 SDA 总线连接了两个上拉电阻，A0 和 A1 接到了低电平。ADT7420 设备地址的高四位为 1001，因此图 9-30 连接时，设备地址为 1001000。

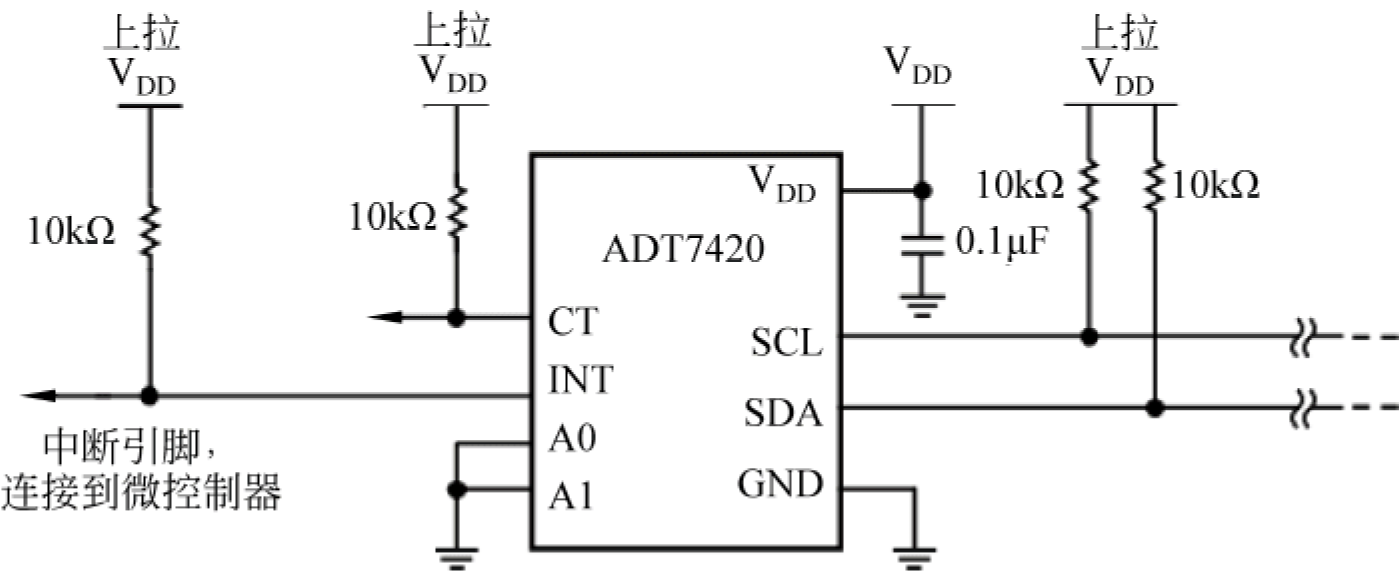


图 9-30　温度传感器 ADT7420 总线连接

ADT7420 内部有多个寄存器用于温度值的存储和传感器配置，如表 9-8 所示，要读写这些寄存器，需要指定寄存器的地址，这和 Flash 的读写时序类似。

表 9-8　ADT7420 寄存器

寄存器地址	描　　述	上电默认值
0x00	温度值最高有效字节	0x00
0x01	温度值最低有效字节	0x00
0x02	状态	0x00
0x03	配置	0x00
0x04	T_{HIGH} 设定点高有效字节	0x20(64℃)
0x05	T_{HIGH} 设定点低有效字节	0x00(64℃)
0x06	T_{LOW} 设定点高有效字节	0x05(10℃)
0x07	T_{LOW} 设定点低有效字节	0x00(10℃)
0x08	T_{CRIT} 设定点高有效字节	0x49(147℃)
0x09	T_{CRIT} 设定点低有效字节	0x80(147℃)

续表

寄存器地址	描　　述	上电默认值
0x0A	T_{HYST} 设定点	0x05(5℃)
0x0B	ID	0xCB
0x2F	软件复位	0xXX

图 9-31 为对 ADT7420 的寄存器进行写的时序，首先发送 I2C 设备地址，方向为写，地址匹配后，主设备发送 8 位的寄存器地址，随后将需要写入该寄存器的数据发送给从设备，以 STOP 结束传输。

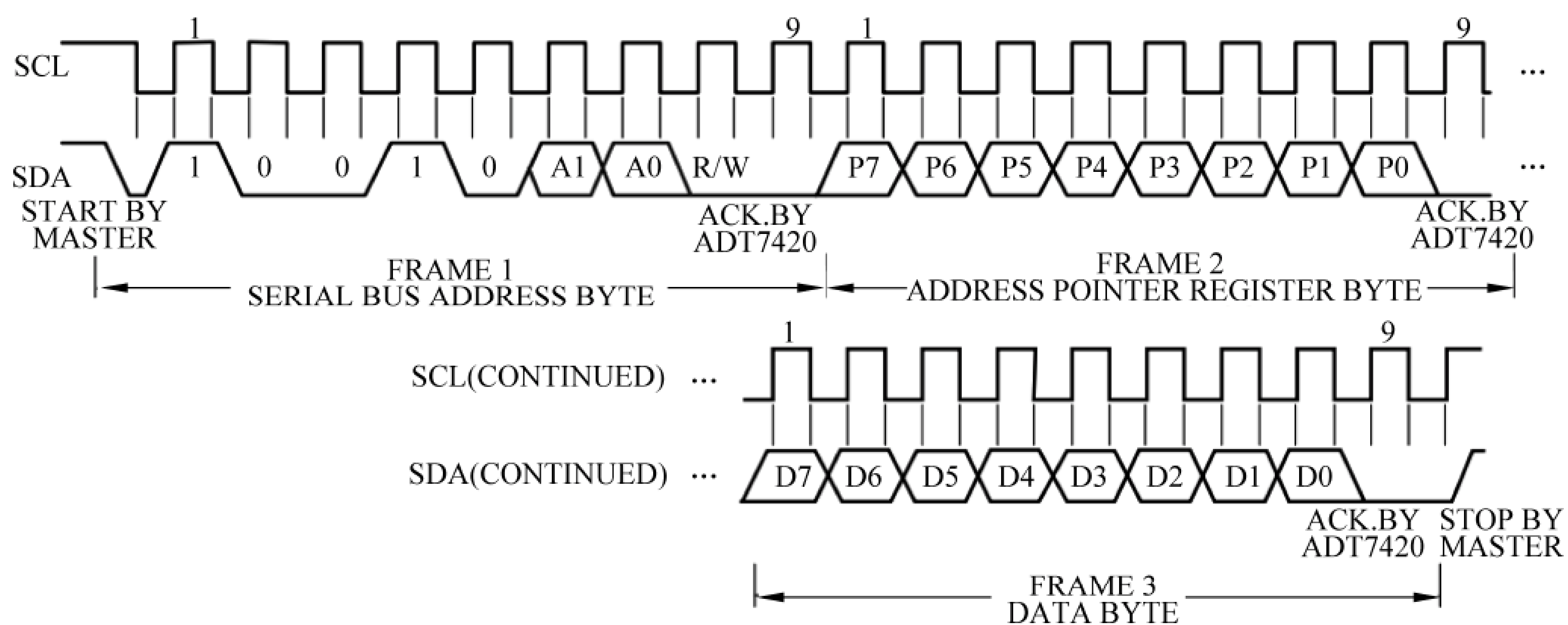


图 9-31　写寄存器时序

图 9-32 为对 ADT7420 的寄存器进行读的时序，首先发送 I2C 设备地址，方向为写，地址匹配后，主设备发送 8 位的寄存器地址，ADT7420 给出响应后，根据寄存器地址准备数据，主设备需要切换方向，重新发送 I2C 地址并将方向切换到读，ADT7420 匹配后将准备好

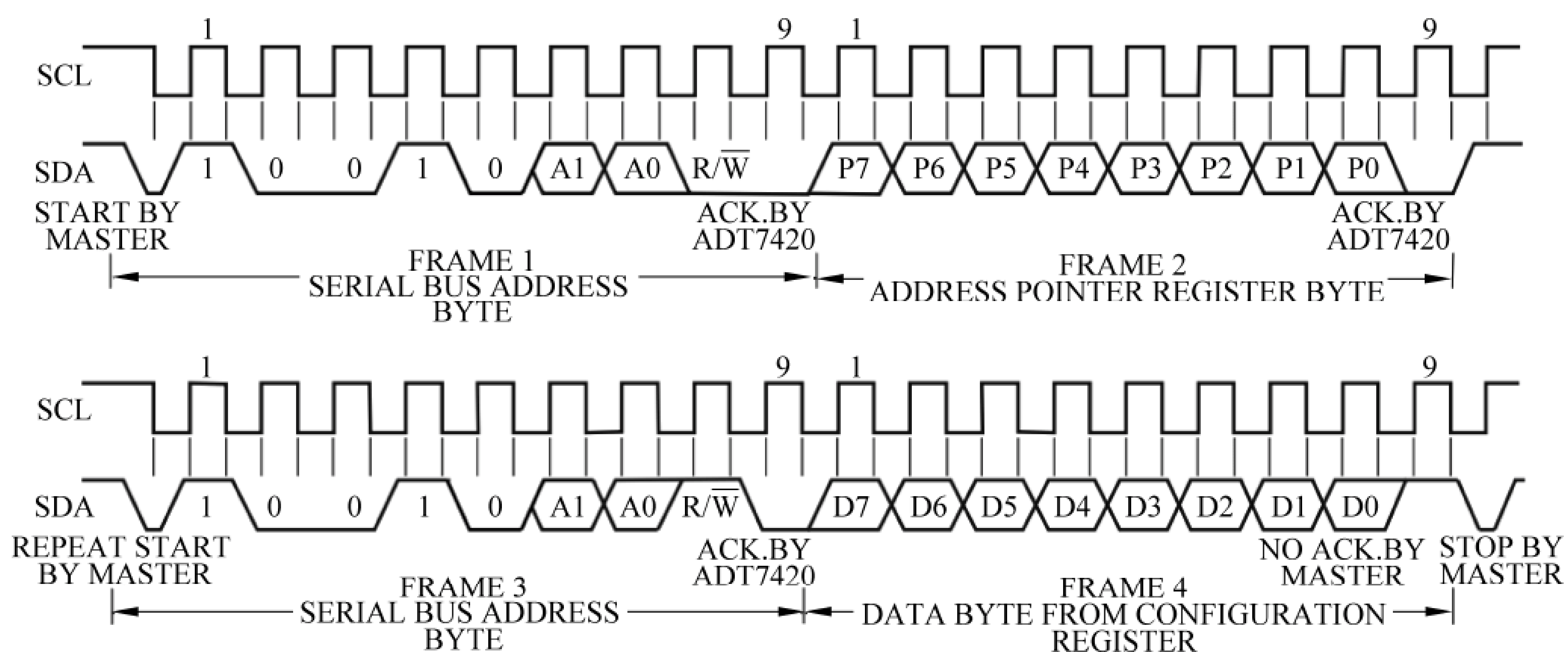


图 9-32　读寄存器时序

的数据发送给主设备，读完最后一个数据后，主设备发送一个 NACK 和 STOP，完成一次数据的读操作。

温度转换的函数如下：

```
float ADT7420_GetTemperature(void)
{
    uint8_t  msbTemp = 0;
    uint8_t  lsbTemp = 0;
    uint8_t  temp    = 0;
    float    tempC   = 0;
    msbTemp = ADT7420_GetRegisterValue(0x00);
    lsbTemp = ADT7420_GetRegisterValue(0x01);
    temp = ((uint16_t)msbTemp << 8) + lsbTemp;
    if(temp & 0x8000)                                //负温度
        tempC = (float)((int32_t)temp - 65536) / 128;
    else
    tempC = (float)temp / 128;
    return tempC;
}
```

其中，读取温度寄存器的函数实现如下：

```
uint8_t ADT7420_GetRegisterValue(uint8_t registerAddress)
{
    uint8_t registerValue = 0;
    I2C_Write(ADT7420_ADDRESS, &registerAddress, 1, 0);
    //设备地址，寄存器地址，写入数据个数，是否发送停止
    I2C_Read(ADT7420_ADDRESS, &registerValue, 1, 1);
    //设备地址，读到的数据值，读数据的个数，是否发送停止位
    return registerValue;
}
```

I2C_Write 将 I2C 设备地址和寄存器地址写到 ADT7420，并不发送 STOP，I2C_Read 将 I2C 设备地址发送到 ADT7420，读取寄存器数据并发送 STOP 结束。

Chapter 10

第 10 章　SPI

【导读】 SPI 是快速同步串行总线，多用于数字外设之间的连接，本章首先介绍了 SPI 总线的概念和时序，然后对 STM32L152 的 SPI 总线控制器的内部结构，寄存器以及发送和接收流程进行了介绍，最后介绍了 CMSIS 提供的典型寄存器操作库函数。针对 SPI 总线时序传输，本章以温度传感器的操作时序为例进行了案例说明。

10.1　SPI 总线概述

SPI(Serial Peripheral Interface)总线是一种同步串行传输总线，允许主机以全双工与外围设备进行高速数据通信，主要用于嵌入式系统短距离通信。SPI 总线由摩托罗拉公司 80 年代后期提出，成为业界标准，但不同公司的处理器的实现细节可能有所不同，主要体现在寄存器定义、数据格式等。SPI 通常为四线制，因此也叫做四线串行总线，以区别单总线、双线和三线串行总线。

四线 SPI 支持全双工通信，采用由一个主设备管理的主从(master-slave)架构，主设备发起读写命令，通过片选信号与多个从设备进行通信。相对于 I2C 总线和 USART 总线，其主要优点为：

- 支持全双工通信。
- 总线驱动性能较好，可以支持 100MHz 以上的高速应用。
- 协议支持 8/16b 字长，可根据应用特点灵活选择字长。
- 硬件连接简单，只需四根信号线(也可支持三线传输)，相比 I2C 不需要仲裁。
- 从设备使用主设备时钟，且由片选选通从设备，无须寻址。

SPI 总线缺点如下：

- 片选选择不同从设备导致多从设备时需要更多的 I/O。
- 没有数据流控制，只能通过降低时钟匹配传输速度。
- 没有从设备接收数据 ACK。
- 典型应用只支持单主控。
- 相比于 RS-232、RS-422、RS-485 和 CAN，SPI 传输距离较短。

SPI 以其简单高效被大多数嵌入式处理器作为标准外设控制器集成到 MCU 中，一些典型外设也采用 SPI 总线接口，如 SD 卡、LCD 显示屏、串行 Flash 存储、RTC 芯片、振动、

压力等传感器。

10.2 SPI 总线控制器架构

10.2.1 接口信号和连接方式

SPI 协议定义四根信号线，分别为：

- SCK(Serial Clock)：串行时钟，作为主设备的输出，从设备的输入。
- MOSI(Master Output, Slave Input)：主设备输出/从设备输入，用于主模式发送数据，从模式接收数据。
- MISO(Master Input, Slave Output)：主设备输入/从设备输出，用于主模式接收数据，从模式发送数据。
- NSS(Slave Select)：从设备片选信号。

其中 MISO 方向为从设备到主设备，其余三个信号均为主设备到从设备。在支持三线 SPI 的控制器中，在三线双向模式下，主设备的 MOSI 和从设备的 MISO 作为双向 I/O 使用。

四线 SPI 主设备和从设备的硬件连接如图 10-1 所示。MOSI 脚相互连接，MISO 脚相互连接。这样，数据在主设备和从设备之间串行地传输。

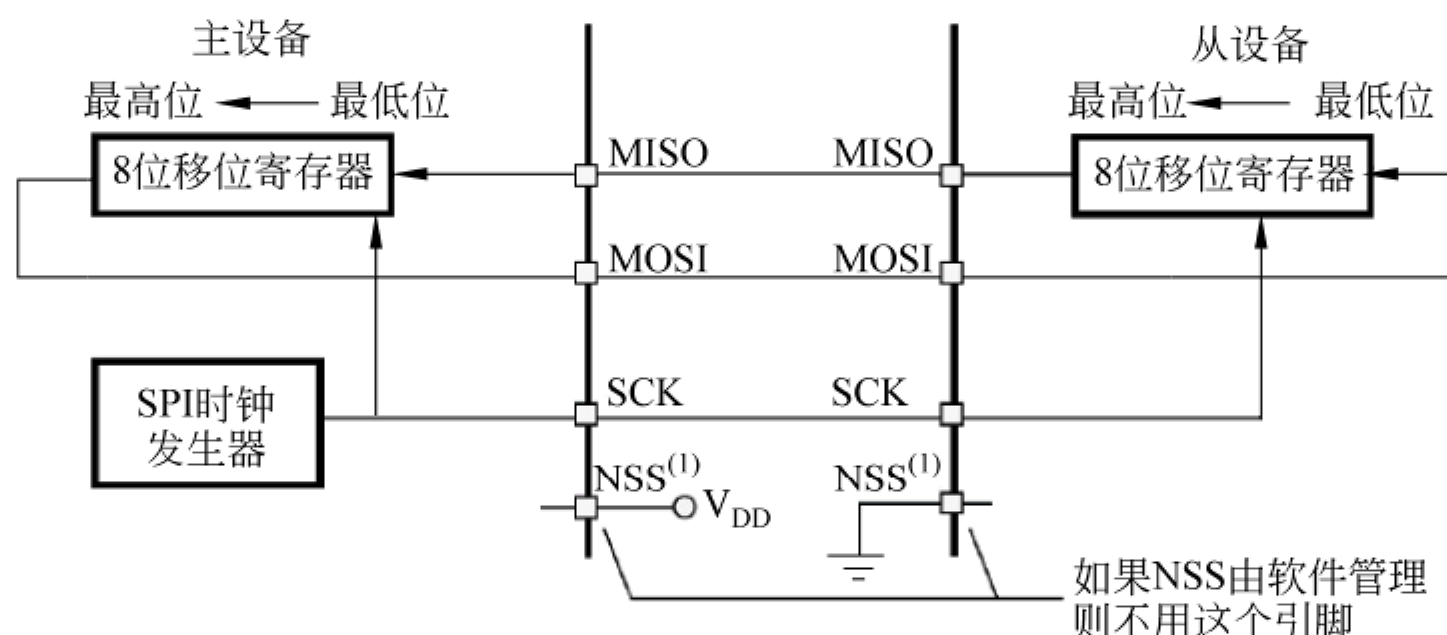


图 10-1 主从 SPI 设备的连接示意图

SPI 规定了两个 SPI 设备之间通信必须由主设备来控制从设备。从设备的时钟由主设备通过 SCK 引脚提供给从设备，从设备本身不能产生或控制 Clock。数据传输由主设备发起，主设备通过 MOSI 脚把数据发送给从设备，从设备通过 MISO 引脚回传数据，这意味全双工通信的数据输出和数据输入是用同一个时钟信号同步控制的。从设备在接收主设备的控制信号前，主设备首先通过 NSS 对从设备进行片选。图 10-1 中，只有两个设备连接，主设备的 NSS 连接到高电平，NSS 对于主设备不起作用，从设备的 NSS 连接到低电平，即一直选通该从设备。

一个主设备通过片选可以控制多个从设备，让主设备可以单独地与特定从设备通信，避免数据线上的冲突。当需要连接多个从设备时，可以通过多片选或者菊花链方式进行连接。

（1）多片选连接方式：从设备的 NSS 引脚可以由主设备的任意一个标准 I/O 引脚来驱动，通常使用多个 I/O 口分别连接到从设备的 NSS 进行控制。如图 10-2 所示，所有从设备的 SCK、MOSI、MISO 都是连在一起，每个从设备都需要单独的片选信号，主设备每次只能选择其中一个从设备进行通信。由于每个设备都需要单独的片选信号，会占用较多的 I/O 资源，可以使用译码器电路或者采用菊花链方式。

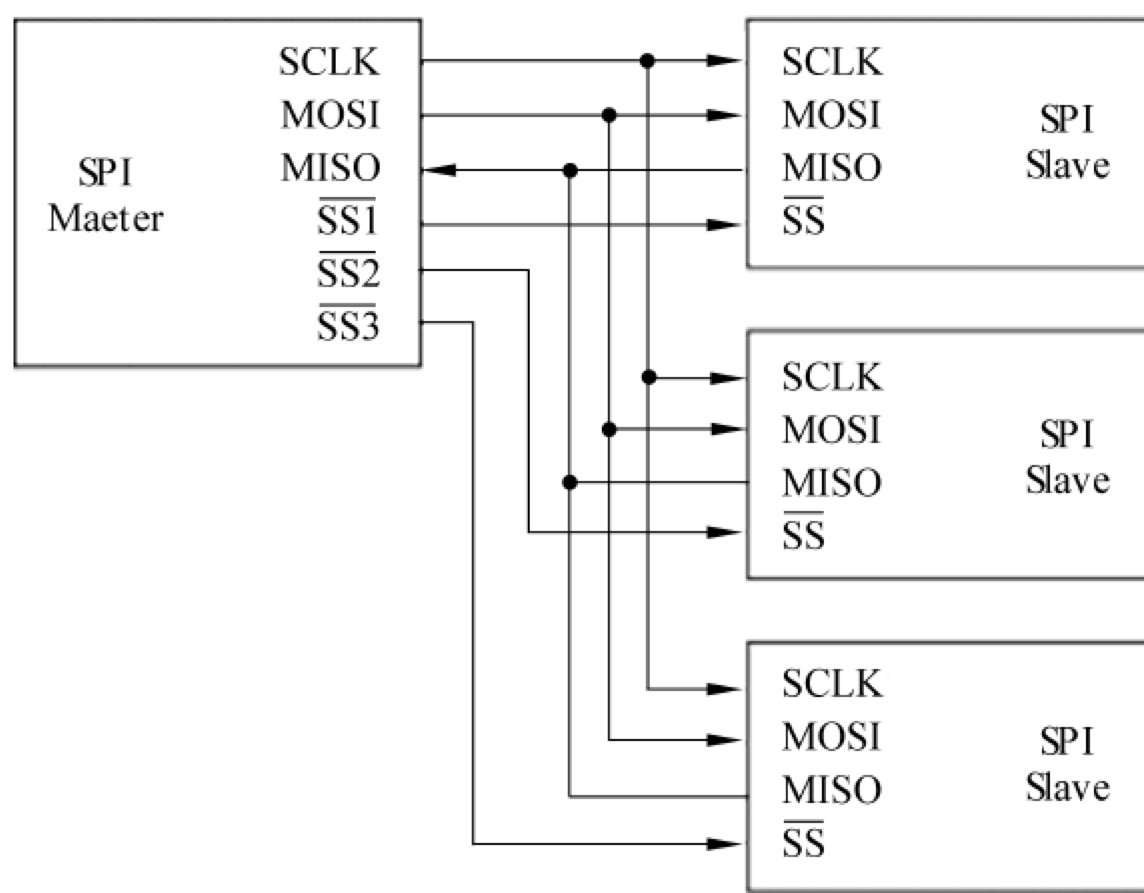


图 10-2　多片选方式控制多个从设备

（2）菊花链连接方式：多片选方式占用较多的 I/O，只用一个 NSS 控制时，可以连接成如图 10-3 所示的电路，不同于图 10-2 的共享 MOSI 和 MISO 总线，菊花链方式下，所有从设备连接到一个 NSS 片选上，主设备 MOSI 连接到第一个从设备，第一个从设备的 MISO 连接到第二个从设备的 MOSI，依次连接，最后一个从设备的 MISO 连接到主设备的

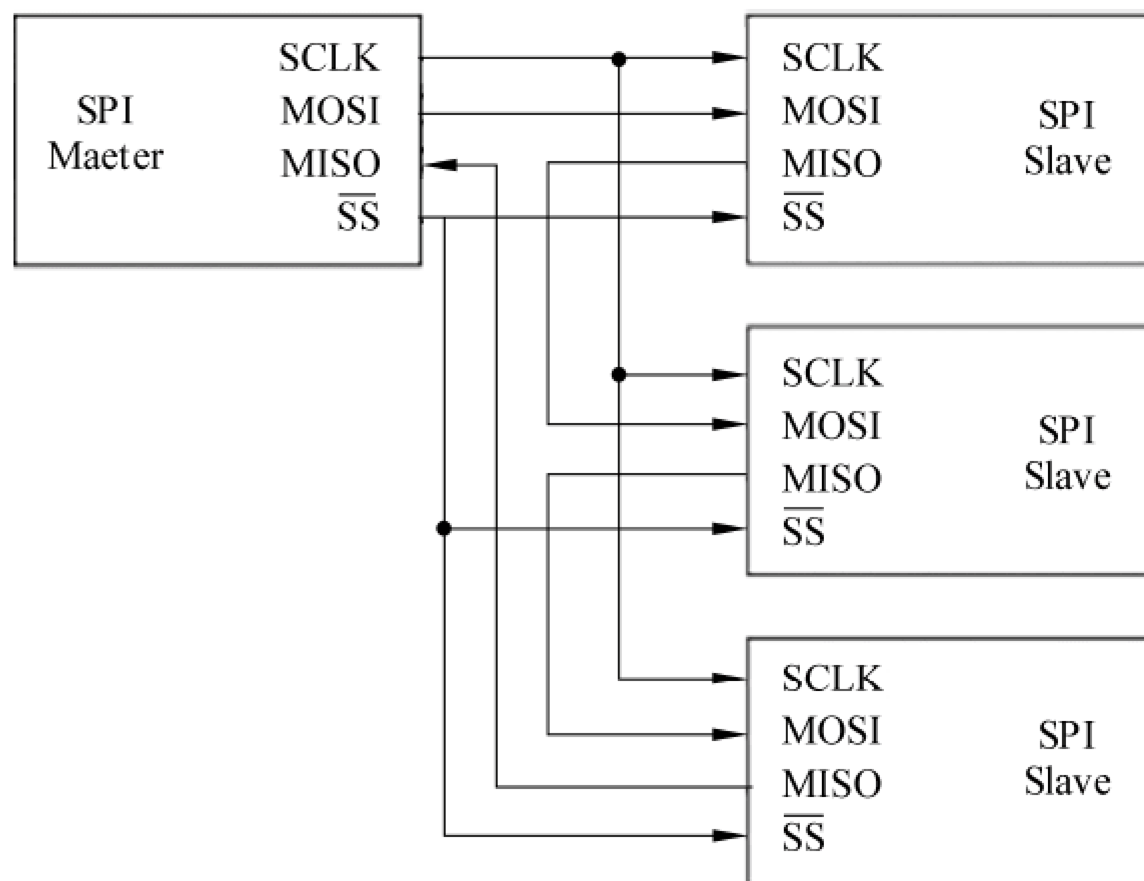


图 10-3　菊花链方式控制多个从设备

MISO,形成一个闭环。数据通过主设备发送,所有的从设备依次接收数据并向下传递。

由于 SPI 是全双工通信,因此,主从设备在数据通信过程中不能只充当一个发送者或者接收者,而是在每个时钟周期内主从设备都会发送并接收一个比特的数据,相当于设备间交换了一比特数据。其内部实现结构如图 10-4 所示,主从设备中均有一个移位寄存器用于存放所要传输的数据,在主设备的时钟控制下同步进行操作。在同一个时钟周期,主设备将自己移位寄存器中的最高位 MSB 通过 MOSI 送出,所有低位数据向高位移动 1 位,从设备将自己的最高位通过 MISO 送出,并将自己移位寄存器中的数据向高位移动 1 位,将 MOSI 上的来自主设备的数据存储到最低位;主设备采集 MISO 上的数据也存储到自己移位寄存器的最低位,这样循环 8 个时钟周期后,主从设备交换了一个字节。需要传输多个字节时,重复上述过程,传输完成后,通过片选信号释放从设备,这样主机的 MOSI 信号将被从设备忽略。单向传输时也保持上述流程,程序不处理从设备接收到的数据即可。

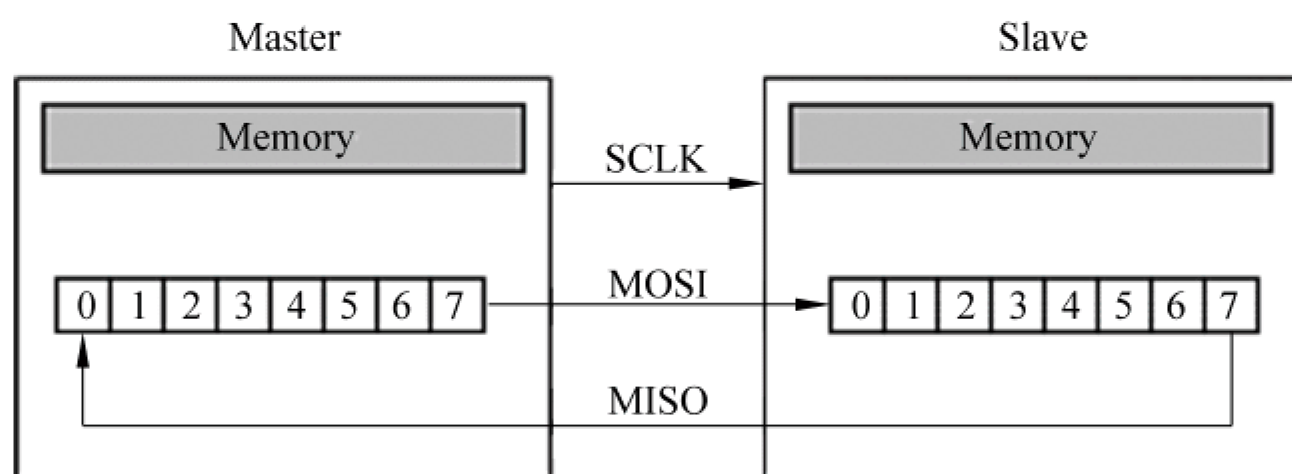


图 10-4　SPI 双向数据传输原理

表 10-1 为数据交换的时序案例。初始状态,主机发送数据 0xAA 到从机,从机发送数据 0x55 到主机,数据都送到了移位寄存器中。当第一个时钟边沿到达时,两个移位寄存器同时进行移位,将高位 MSB 的数据发送到数据线,在第二个时钟边沿到达时,将数据线 MOSI 和 MISO 同时存储到各自的移位寄存器,实现了一个数据位的交换。8 个时钟周期后,完成一个字节的数据交换。

表 10-1　数据交换时序

时钟脉冲	主机移位寄存器	从机移位寄存器	MISO	MOSI
0	10101010	01010101	0	0
1 上	0101010x	1010101x	0	1
1 下	01010100	10101011	0	1
2 上	1010100x	0101011x	1	0
2 下	10101001	01010110	1	0
3 上	0101001x	1010110x	0	1
3 下	01010010	10101101	0	1
4 上	1010010x	0101101x	1	0

续表

时钟脉冲	主机移位寄存器	从机移位寄存器	MISO	MOSI
4 下	10100101	01011010	1	0
5 上	0100101x	1011010x	0	1
5 下	01001010	10110101	0	1
6 上	1001010x	0110101x	1	0
6 下	10010101	01101010	1	0
7 上	0010101x	1101010x	0	1
7 下	00101010	11010101	0	1
8 上	0101010x	1010101x	1	0
8 下	01010101	10101010	1	0

10.2.2　传输模式和时序

在数据传输的过程中，主从设备必须在下一次数据传输之前将接收到的数据采样保存。SPI 的数据采样的时机和时钟的相位可以由两个控制信号 CPOL 和 CPHA 灵活进行配置。

CPOL(Clock Polarity)：决定在没有数据传输时时钟的空闲状态电平是高电平还是低电平，该信号为 1 时 SCK 引脚在空闲状态保持高电平，为 0 时 SCK 引脚在空闲状态保持低电平。

CPHA(Clock Phase)：定义 SPI 数据采样的时机，该信号为 1 时数据采样发生在时钟 SCK 的第二个边沿(CPOL 位为 0 时就是下降沿，CPOL 位为 1 时就是上升沿)，为 0 时数据采样发生在时钟 SCK 的第一个边沿(CPOL 位为 0 时就是上升沿，CPOL 位为 1 时就是下降沿)。

根据 CPOL 和 CPHA 的组合，将 SPI 可以分成 4 种传输模式，如表 10-2 所示，分别记为 SPI0、SPI1、SPI2 和 SPI3，其具体时序如图 10-5 所示。主从设备进行 SPI 通信时，要确保它们的传输模式设置相同。

表 10-2　四种传输模式

模　式	CPOL	CPHA
SPI0	0	0
SPI1	0	1
SPI2	1	0
SPI3	1	1

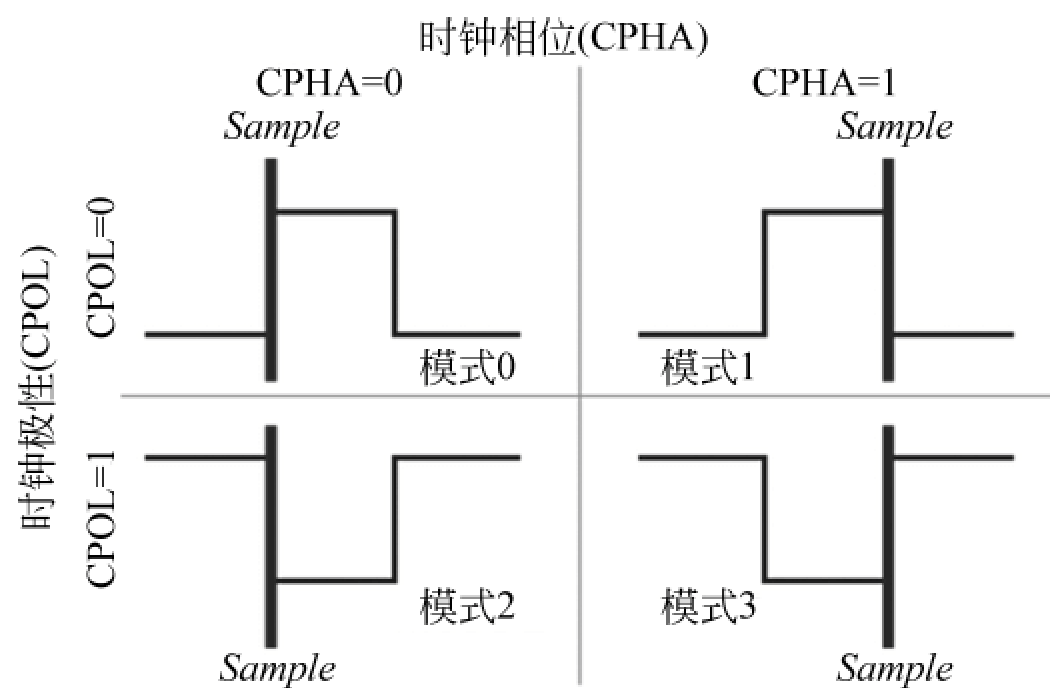


图 10-5 四种时序模式

图 10-6 为 SPI 传输的 4 种模式的时序。

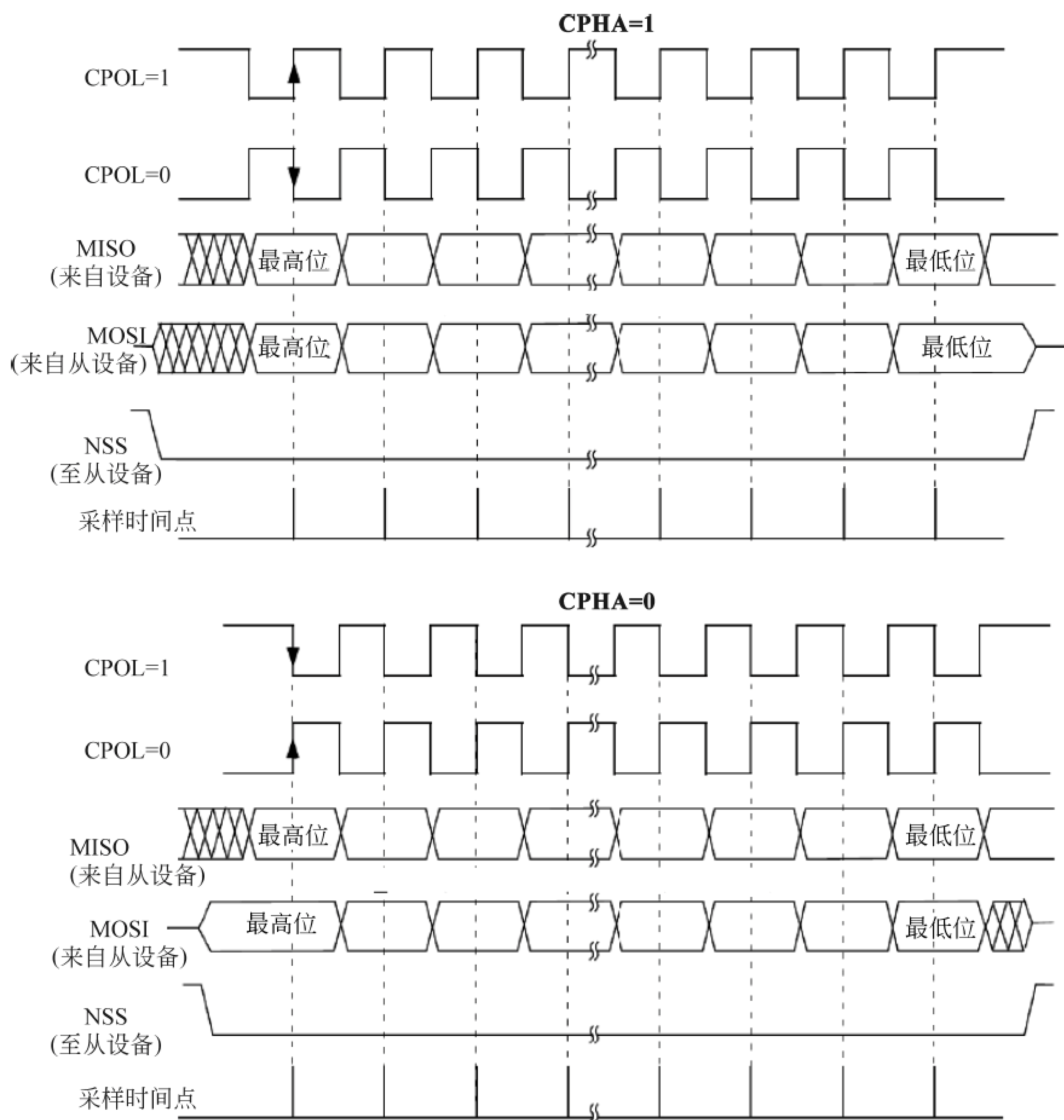


图 10-6 SPI 4 种模式下的采样时序

CPHA＝1 时：

- 在 SCK 第二个时钟边沿采样和锁存数据。
- 在 SCK 第三个时钟边沿，将上个时钟边沿锁存的数据写入移位寄存器。
- 以此类推，数据在偶数边沿锁存，在奇数边沿写入移位寄存器。
- 经过 8/16 个时钟边沿后，串行传输的数据全部写入移位寄存器，完成主从设备的数据交换。

CPHA＝0 时：

- SCK 的第一个时钟边沿采样和锁存数据。
- 在 SCK 的第二个时钟边沿，上个时钟边沿锁存的数据写入移位寄存器。
- 以此类推，数据在奇数边沿锁存，在偶数边沿写入移位寄存器。
- 经过 8/16 个时钟边沿后，串行传输的数据全部写入移位寄存器，完成主从设备的数据交换。

10.2.3 STM32L15x SPI 总线控制器

STM32L15x 系列微控制器有两个独立的 SPI 控制器，SPI1 连接在 APB2 总线上，SPI2 连接在 APB1 总线上，对于 STM32L15x 的 APB1 和 APB2 总线频率最高都支持到 32MHz，因此两个 SPI 控制器在最高通信速率上没有区别。

STM32L15x SPI 总线控制器支持 4 线和 3 线连接，8/16 位可选数据帧，支持多主模式，最大时钟频率为 APB 总线频率的 1/2，同时支持 8 种预分频系数。主从模式下 NSS 可以由软件或硬件管理，主从模式可动态切换，支持硬件 CRC，可编程数据传输顺序、DMA 传输以及专用的中断和总线状态标志位。其内部结构如图 10-7 所示。

如图 10-7 所示，SPI 控制器由接收和发送缓冲区，移位寄存器、波特率发生器、主控电路、通信电路以及控制和状态寄存器构成。波特率发生器用于控制时钟 SCK，其主要由 CR1 寄存器的分频因子 BR 和极性相、位控制信号 CPOL、CPHA 进行配置。主控电路用于确定 SPI 的双工/单工，3/4 线连接模式等，控制移位寄存器的时序，移位寄存器的数据输出次序（高位优先还是低位优先）由 CR1 寄存器的 LSBFIRST 决定。接收缓冲区用于存储从 MISO 发来的数据，发送缓冲区用于存储向 MOSI 总线发送的数据。通信电路用于管理片选信号 NSS、主从模式选择和 CRC 校验、中断和总线状态等。

STM32L15x 的每个 SPI 的 NSS 可以配置为输入，也可以配置为输出，可以通过 CR2 寄存器的 SSOE 控制。配置为输入时，NSS 的电平信号用于控制 SPI 控制器自己，配置为输出时，NSS 的信号发送给从设备进行片选，当配置为输出时，只有一个设备为主设备，其余设备均为从设备，不支持多主设备工作。SPI 控制器的外部接口 NSS 引脚连接到 SPI 控制器内部时的电路结构如图 10-8 所示，实际控制 SPI 片选的是内部 NSS，其有两个来源，分别为外部 NSS 引脚和内部寄存器 SSI 位，CR1 寄存器的 SSM 用于控制内部 NSS 的信号来源。

当 SSM 选通 SSI 作为内部 NSS 信号来源时，称为软件 NSS 管理模式，此时外部 NSS

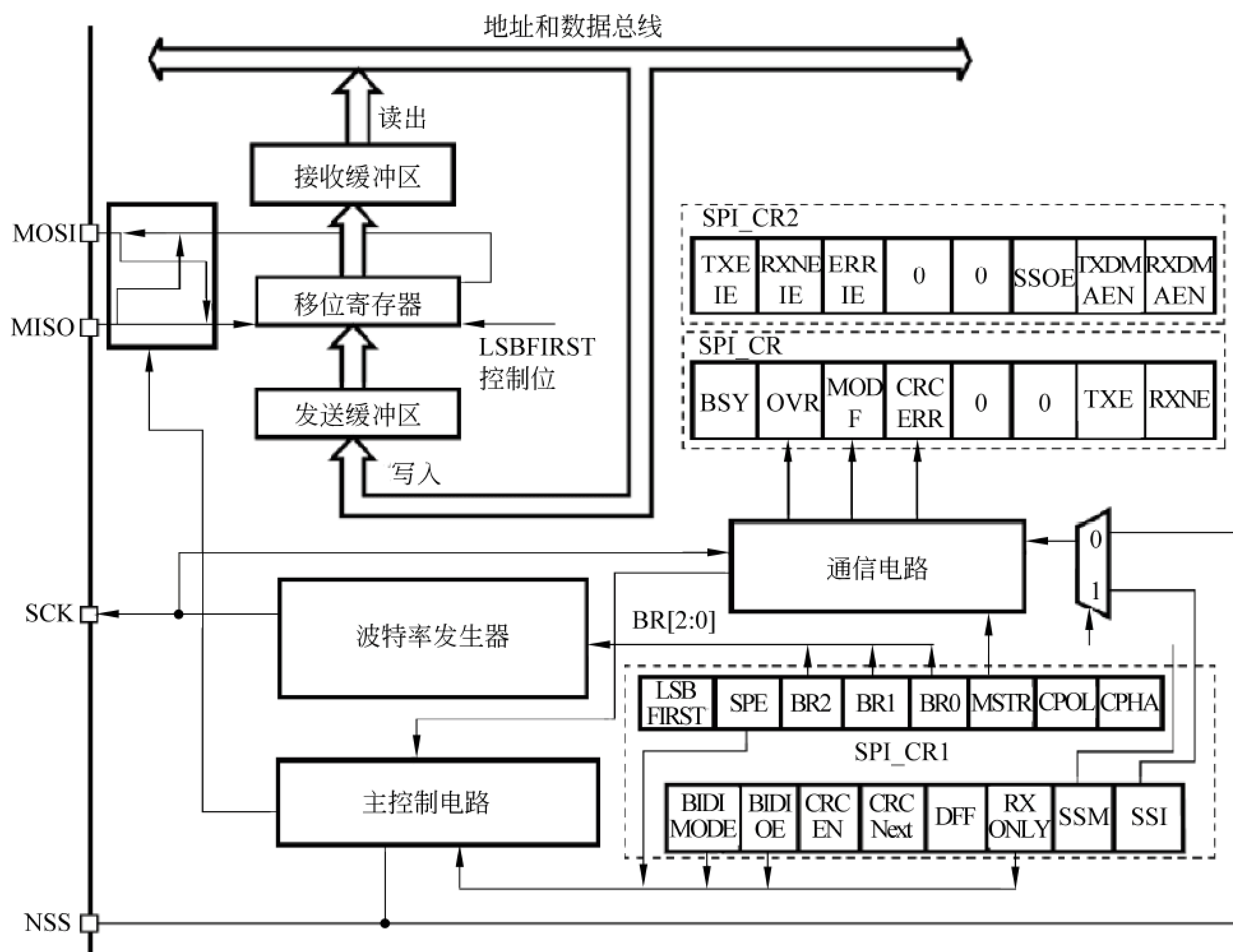


图 10-7　SPI 控制器内部结构图

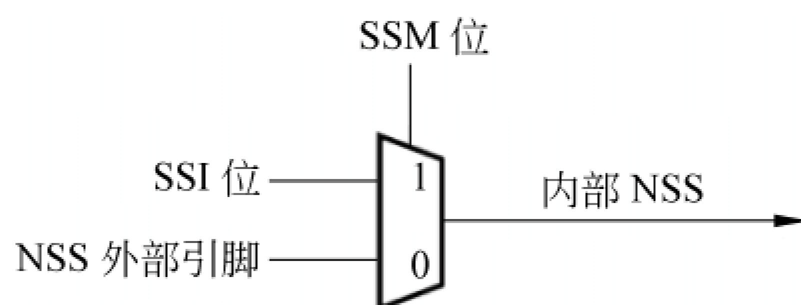


图 10-8　NSS 引脚选择内部电路

引脚无效，可以作为普通 GPIO 使用。当 SPI 工作在主模式时，SSI 需置为 1，当 SPI 工作在从设备模式下时，SSI 需配置为 0。例如，将微控制器集成的 SPI1 和 SPI2 控制器分别作为主设备和从设备连接进行数据传输，此时可配置 SPI1 SSI 位为 1，SPI2 的 SSI 为 0，无需连接 NSS 引脚。一般情况下，外设多为从设备，此时将 SPI 配置为主设备、软件管理模式，NSS 外部引脚作为 GPIO 控制从设备的片选，需要通过额外的 GPIO 控制对从设备进行片选。

当 SSM 选通 NSS 外部引脚作为内部 NSS 信号源时，称为 NSS 硬件管理模式。硬件管理模式下，当 NSS 配置为输出时，主设备一旦开始数据传输，自动将 NSS 置为 0，此时片选连接到 NSS 的其他设备均成为从设备接收数据(注意，NSS 信号不会自动变为 1)。硬件模式只用于多主设备下，当一个主设备发送数据时，它会自动拉低 NSS 信号，以通知所有其他

的设备它是主设备，如果它不能拉低 NSS，这意味着总线上有另外一个主设备在通信，这时将产生一个硬件失败错误(Hard Fault)。

10.3 SPI 寄存器说明

SPI 控制器的寄存器如表 10-3 所示。

表 10-3 SPI 寄存器列表

寄存器名称	偏移量	功　能	复位值
控制寄存器 1(SPI_CR1)	0x00	控制寄存器 1，用于配置帧格式、波特率、时钟以及三线、四线模式	0x0000 00C0
控制寄存器 2(SPI_CR2)	0x04	控制寄存器 2，中断和 DMA 管理	0x0000 0000
状态寄存器(SPI_SR)	0x08	状态寄存器，发送寄存器空、接收寄存器空等状态控制	0x0000 0000
数据寄存器(SPI_DR)	0x0C	发送和接收数据	0x0000 0000
CRC 多项式寄存器(SPI_CRC)	0x10	校验方法选择	0x0000 XXXX
RX CRC 寄存器(SPI_RXCRCR)	0x14	CRC 校验值	0x0000 0000
TX CRC 寄存器(SPI_TXCRCR)	0x18	CRC 校验值	0x0000 0000

1. SPI 控制寄存器 1(SPI_CR1)

控制寄存器 SPI_CR1 的有效域定义如图 10-9 所示。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BIDI MODE	BIDI OE	CRC EN	CRC NEXT	DFF	RX ONLY	SSM	SSI	LSB FIRST	SPE	BR [2:0]			MSTR	CPOL	CPHA
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 10-9 控制寄存器 CR1

BIDIMODE(Bidirectional Mode Enable)：单线双向数据模式使能域，用于配置 3 线/4 线 SPI 模式，0 表示双线单向模式，即全双工，1 表示单线双向模式，即半双工。

BIDIOE(Bidirectional Output Enable)：双向模式下的输出使能，和 BIDIMODE 位一起决定在单线双向模式下数据的输出方向，置为 0 时 SPI 为接收数据，置为 1 时发送数据。单线双向模式下，主设备 MOSI 连接到从设备 MISO 引脚，通过控制 BIDIOE 实现输入和输出的切换。

CRCEN(CRC Enable)：硬件 CRC 校验使能，置为 1 启用 CRC 计算。配置该位时需关闭 SPI(SPE 置为 0)，且该位只能在全双工模式下使用。

CRCNEXT(Transmit CRC next)：下一个发送 CRC，该位置为 1 时，SPI 将 CRC 寄存器中值通过移位寄存器发出，置为 0 时将发送缓冲区的数据发出。在启用 CRC 的情况下，当发送完最后一个数据后，将该位置 1，SPI 控制器自动发送 CRC 校验数据。

DFF(Data Frame Format)：数据帧格式域，该位置 0 表示使用 8 位数据帧格式进行发送/接收；置 1 表示使用 16 位数据帧格式进行发送/接收。配置该位时需要关闭 SPI。

RXONLY(Receive Only)：只接收，该位和 BIDIMODE 位一起决定在“双线单向”模式下的传输方向。置 0 表示全双工，置 1 表示只接收不发送。

SSM(Software Slave Management)：软件从设备管理，用于控制内部 NSS 信号的来源，置 1 时，启用软件从设备管理，内部 NSS 引脚上的电平由寄存器中 SSI 位的值决定。

SSI(Internal Salve Select)：内部从设备选择，该位只在 SSM 位为 1 时有效，用于配置内部 NSS 引脚的电平。

LSBFIRST：帧格式控制，0 表示先发送 MSB，1 表示先发送 LSB。

SPE(SPI Enable)：SPI 使能，置 0 禁止 SPI 设备，置 1 开启 SPI 设备。

BR[2：0](Buadrate Control)：波特率控制，三个二进制编码 000～111 分别表示 SPI 的 CLK 为 APB 总线时钟的 2、4、8、16、32、64、128、256 分频。

MSTR(Master Selection)：主设备选择，用于配置 SPI 工作在主设备模式(置为 1)还是从设备模式(置为 0)。

CPOL(Clock polarity)：时钟极性，和 CPHA 位组合用于选择 SPI 时序模式，该位为 0 表示空闲状态时 SCK 保持低电平，为 1 表示空闲状态时 SCK 保持高电平。

CPHA(Clock Phase)：时钟相位，该位置 0 表示数据采样从第一个时钟边沿开始；置 1 表示数据采样从第二个时钟边沿开始。

2. SPI 控制寄存器 2(SPI_CR2)

控制寄存器 SPI_CR2 的有效域定义如图 10-10 所示。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								TXEIE	RXNEIE	ERRIE	FRF	Res.	*SSOE*	TXDMAEN	RXDMAEN
								rw	rw	rw	rw		rw	rw	rw

图 10-10 控制寄存器 CR2

TXEIE(Tx Buffer Empty Interrupt Enable)：发送缓冲区空中断使能，用于配置 SPI 数据传输完成中断，置 0 表示禁止 TXE 中断，置 1 表示允许 TXE 中断。

RXNEIE(Rx Buffer Not Empty Interrupt Enable)：接收缓冲区非空中断使能，用于配置 SPI 数据接收中断，置 0 表示禁止 RXNE 中断，置 1 表示允许 RXNE 中断。

ERRIE(Error Interrupt Enable)：错误中断使能，用于控制当 SPI 传输出现校验错误、溢出、模式错误等时是否产生中断，0 禁止，1 允许。

SSOE(NSS Output Enable)：NSS 引脚的输入输出控制，0 表示 NSS 引脚为输入，1 表示 NSS 引脚为输出。

TXDMAEN(Tx DMA Enable)：发送缓冲区 DMA 使能，当该位被置 1 时，SPI_SR 寄存器中的发送完成标志 TXE 一旦被置 1 就发出 DMA 请求；该位置 0 不启用 DMA 传输。

RXDMAEN(Rx DMA Enable)：接收缓冲区 DMA 使能，当该位被置 1 时，SPI_SR 寄

存器中的 RXNE 标志一旦被置 1 就发出 DMA 请求；该位置 0 不启用 DMA 传输。

3. SPI 状态寄存器(SPI_SR)

控制寄存器 SPI_SR 的有效域定义如图 10-11 所示，包括：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved							FRE	BSY	OVR	MODF	CRC ERR	UDR	CHSIDE	TXE	RXNE
							r	r	r	r	rc_w0	r	r	r	r

图 10-11 状态寄存器 SR

BSY：忙标志，由硬件控制，该位为 0 表示 SPI 不忙，为 1 表示 SPI 正忙于通信，或者发送缓冲非空。SPI 开始传输时 BSY 被自动置 1，当传输结束、关闭 SPI 时被置 0，如果不是连续通信，在每个数据(8 位或 16 位)传输完成后之间，BSY 标志被自动置 0；当通信是连续时，主模式下 BSY 在整个传输期间保持为高电平，从模式下在每个数据传输完成之后被自动置 0，下一个数据开始时自动置 1。

OVR：溢出标志，当主设备已经发送了数据字节，而从设备还没有清除前一个数据字节产生的 RXNE 时，即为溢出错误。该位为 1 表示出现溢出错误，发生溢出错误时硬件自动置 1，需要软件进行清除。

MODF(Mode Fault)：主模式失效错误，该位为 1 表示出现主模式被迫变更为从模式，发生模式错误时硬件自动置 1，需要软件进行清除。

CRCERR(CRC Error)：CRC 错误标志，该位为 1 表示收到的 CRC 值和 SPI_RXCRCR 寄存器中的值不匹配。该位由硬件置位，由软件写 0 复位。

TXE(Tx Empty)：发送缓冲为空，0 表示发送缓冲非空；1 表示发送缓冲为空，可以写下一个待发送的数据进入缓冲器中。当写入 SPI_DR 时，TXE 标志被清除。

RXNE(Rx Not Empty)：接收缓冲非空，0 表示接收缓冲为空；1 表示在接收缓冲器中包含有效的接收数据。读 SPI 数据寄存器可以清除此标志。

FRE、UDR 和 CHSIDE 三个域在 SPI 模式下无效。

4. SPI 数据寄存器(SPI_DR)

数据寄存器如图 10-12 所示，DR 是一个 16 位寄存器，用于存储待发送或者已经收到的数据，数据寄存器 DR 实际对应两个缓冲区：一个用于写的发送缓冲和另外一个用于读的接收缓冲。写操作将数据写到发送缓冲区；读操作将返回接收缓冲区里的数据。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DR[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 10-12 数据寄存器 DR

寄存器 SPI_CR1 的 DFF 位对数据帧格式的选择，DFF 为 0 时数据帧格式为 8 位，发送和接收缓冲器只会用到 SPI_DR[7：0]，其余位为 0。DFF 为 1 时，缓冲器使用整个数据寄存器 SPI_DR[15：0]。

5. SPI CRC 多项式寄存器(SPI_CRCPR)

CRC 多项式寄存器的有效域定义如图 10-13 所示,该寄存器包含了 CRC 计算时用到的多项式。其复位值为 0x0007,根据应用可以设置其他数值。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CRCPOLY[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 10-13　CRC 计算多项式寄存器

6. SPI Rx CRC 寄存器(SPI_RXCRCR)

接收数据 CRC 寄存器的有效域定义如图 10-14 所示,RXCRC[15:0]:接收 CRC 寄存器,在启用 CRC 计算时,RXCRC[15:0]中包含了依据收到的字节计算的 CRC 数值。当在 SPI_CR1 的 CRCEN 位写入'1'时,该寄存器被复位。CRC 计算使用 SPI_CRCPR 中的多项式。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RXCRC[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

图 10-14　接收数据 CRC 计算值

当数据帧格式被设置为 8 位时,仅低 8 位参与计算,并且按照 CRC8 的方法进行;当数据帧格式为 16 位时,寄存器中的所有 16 位都参与计算,并且按照 CRC16 的标准。

7. SPI Tx CRC 寄存器(SPI_TXCRCR)

发送数据 CRC 寄存器的有效域定义如图 10-15 所示,TxCRC[15:0]:发送 CRC 寄存器,在启用 CRC 计算时,TXCRC[15:0]中包含了依据将要发送的字节计算的 CRC 数值。当在 SPI_CR1 中的 CRCEN 位写入'1'时,该寄存器被复位。CRC 计算使用 SPI_CRCPR 中的多项式。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TXCRC[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

图 10-15　发送数据 CRC 计算值

10.4　SPI 通信流程

在进行 SPI 通信之前,首先要确定 SPI 的主从工作模式,一般情况下,MCU 被配置成主设备模式和其他外设进行数据通信,如果 MCU 作为从设备连接到其他设备上,需要将 MCU 的 SPI 配置为从设备模式。无论是 MCU 配置成主模式还是从模式,都需要根据所连接的从设备或主设备的配置进行参数配置。

10.4.1 SPI 双工通信模式配置

1. 从模式配置

在从设备模式下，SCK 引脚用于接收从主设备来的串行时钟，SPI_CR1 寄存器中 BR[2：0]的设置不影响数据传输速率。从设备模式的配置步骤如下：

- 设置 DFF 位以定义数据帧格式为 8 位或 16 位。
- 根据主设备的 CPOL 和 CPHA 时序要求，选择相同的 CPOL 和 CPHA 配置来定义数据传输和串行时钟之间的相位关系。
- 配置 SPI_CR1 寄存器中的 LSBFIRST 位确保帧格式与主设备相同。
- 配置 I/O 方向，MOSI 为输入，MISO 为输出，SPI_CR2 中的 SSOE 置为 0(NSS 引脚为输入)。
- 选择 NSS 的模式，若 NSS 连接到 GND 或主设备的 NSS 或 I/O 引脚，则选用硬件模式。若 NSS 未连接，则配置 NSS 软件模式，设置 SPI_CR1 寄存器中的 SSM 位并清除 SSI 位。
- 如果启用接收和发送中断，配置 SPI_CR2 寄存器的 TXEIE 和 RXNEIE，若启用 DMA 传输中断，配置 TXDMAIE 和 RXDMAIE。
- 清除 SPI_CR1 寄存器 MSTR 位、设置 SPE 位，启动 SPI 工作。

2. 主模式配置

在主设备模式下，主设备在 SCK 脚输出串行时钟。

- 通过 SPI_CR1 寄存器的 BR[2：0]位定义串行时钟波特率。
- 选择 CPOL 和 CPHA 位，定义数据传输和串行时钟间的相位关系。
- 设置 DFF 位来定义 8 位或 16 位数据帧格式。
- 配置 SPI_CR1 寄存器的 LSBFIRST 位定义帧格式。
- 配置引脚 I/O，MOSI 配置为输出，MISO 配置为输入。
- 配置 NSS 引脚的输入输出模式，输入硬件模式下，NSS 脚连接到高电平；输入软件模式下，需设置 SPI_CR1 寄存器的 SSM 位和 SSI 位；输出模式下，置 SSOE 位为 1。
- 设置 MSTR 位和 SPE 位，启动 SPI 传输。

3. 主从模式下的数据发送与接收流程

主模式下，当写入数据到 SPI_DR 寄存器(发送缓冲器)后，传输开始；在发送第一个数据位时，数据被并行地从发送缓冲器传送到 8 位的移位寄存器中，然后按顺序被串行地移位送到 MOSI 引脚上；MSB 在先还是 LSB 在先，取决于 SPI_CR1 寄存器中的 LSBFIRST 位的设置。数据从发送缓冲器传输到移位寄存器时 TXE 标志将被置位，如果设置了 SPI_CR1 寄存器中的 TXEIE 位，将产生中断。与此同时，在 MISO 引脚上接收到的数据，按顺序被串行地移位进入 8 位的移位寄存器中。

从模式下，当从设备接收到时钟信号并且第一个数据位出现在它的 MOSI 时，数据通信开始，MOSI 上传输第一个数据位时，发送缓冲器中的数据被并行地传送到移位寄存器，

随后 MOSI 的数据位依次移动进入移位寄存器，移位寄存器中的数据依次被发送到 MISO 引脚上；在 SPI 主设备开始数据传输之前，从设备需在发送寄存器中提前写入要发送的数据。当发送缓冲器中的数据传输到移位寄存器时，SPI_SP 寄存器的 TXE 标志被置 1，写入 SPI_DR 寄存器 TXE 被清 0。如果设置了 SPI_CR2 寄存器的 TXEIE 位，将会产生中断。

无论工作在何种模式，SPI 控制器在最后一个采样时钟边沿后，将移位寄存器中的数据传送到接收缓冲器，SPI_SR 寄存器中的 RXNE 标志被置 1，表明接收数据已就绪。读 SPI_DR 寄存器时，SPI 设备返回这个接收缓冲器的数值并将 RXNE 位置 0。如果设置了 SPI_CR2 寄存器中的 RXNEIE 位，则产生中断。

在发送和接收的过程中，SPI 处于通信状态是 BSY 标志被置 1，可以通过 BSY 信号判断传输是否完成。从模式下，一旦传输开始，如果下一个将发送的数据被放进了发送缓冲器，就可以维持一个连续的传输流，如图 10-16 所示，配置 LSB 优先，CPOL＝CPHA＝1，主设备发送 0xF1、0xF2、0xF3，从设备发送 0xA1、0xA2、0xA3。在将数据 0xF1 写入到数据寄存器后，TXE 标志被自动置 0，SPI 检测 SCK 时钟沿，在第一个下降沿，主模式下将会立即将 0xF1 写入发送缓冲器并将最低位发送到 MOSI 总线，BSY 标志被置 1，此时 SPI_DR 寄存器可以接收下一个将要被发送的数据，TXE 标志被自动置 1。在第一个上升沿，SPI 锁存 MISO 总线的 0xA1 的最低位电平信号，在第二个下降沿时，将 MISO 的信号写入移位寄存器最低位并在 MOSI 总线上送出 0xF1 的第二位，移位寄存器右移 1 位。在第 8 个时钟周期后，一个字节传输完成，BSY 标志被自动置 0，数据已被保存到接收缓冲区，RXNE 标志被自动置 1。第 9 个时钟周期，SPI 开始进行数据 0xF2 的发送和 0xA2 的接收。写发送缓冲

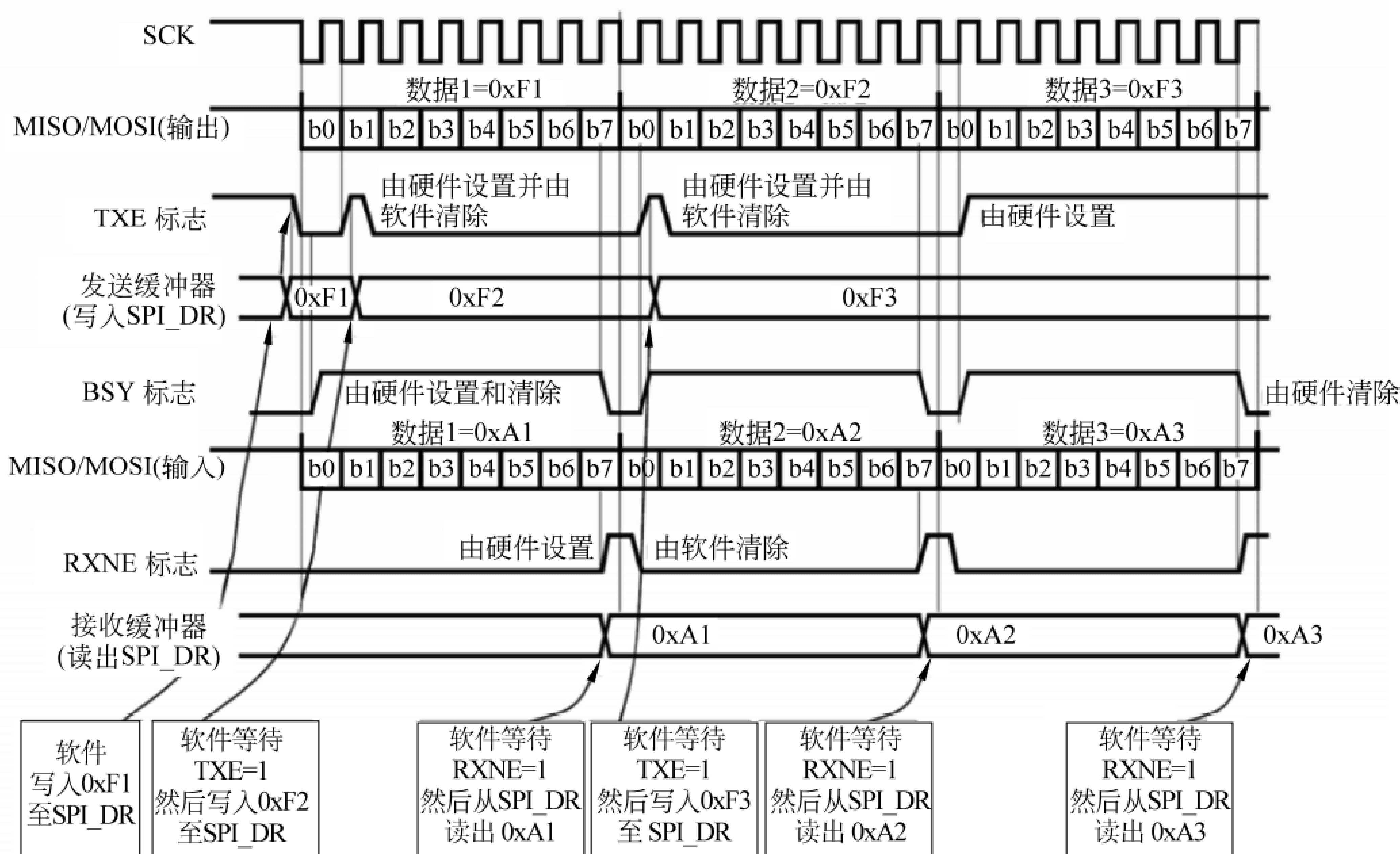


图 10-16　全双工从模式数据发送

器之前，需确认 TXE 标志应该为 1，否则新的数据会覆盖已经在发送缓冲器中的数据。主模式的时序和从模式类似，如图 10-17 所示，不同之处在于 BSY 信号在连续进行数据发送时一直保持高电平，直到所有数据发送结束，而从模式下 BSY 信号在每接收到一个主设备数据后被置为 0，下一个数据到达时又被置为 1。

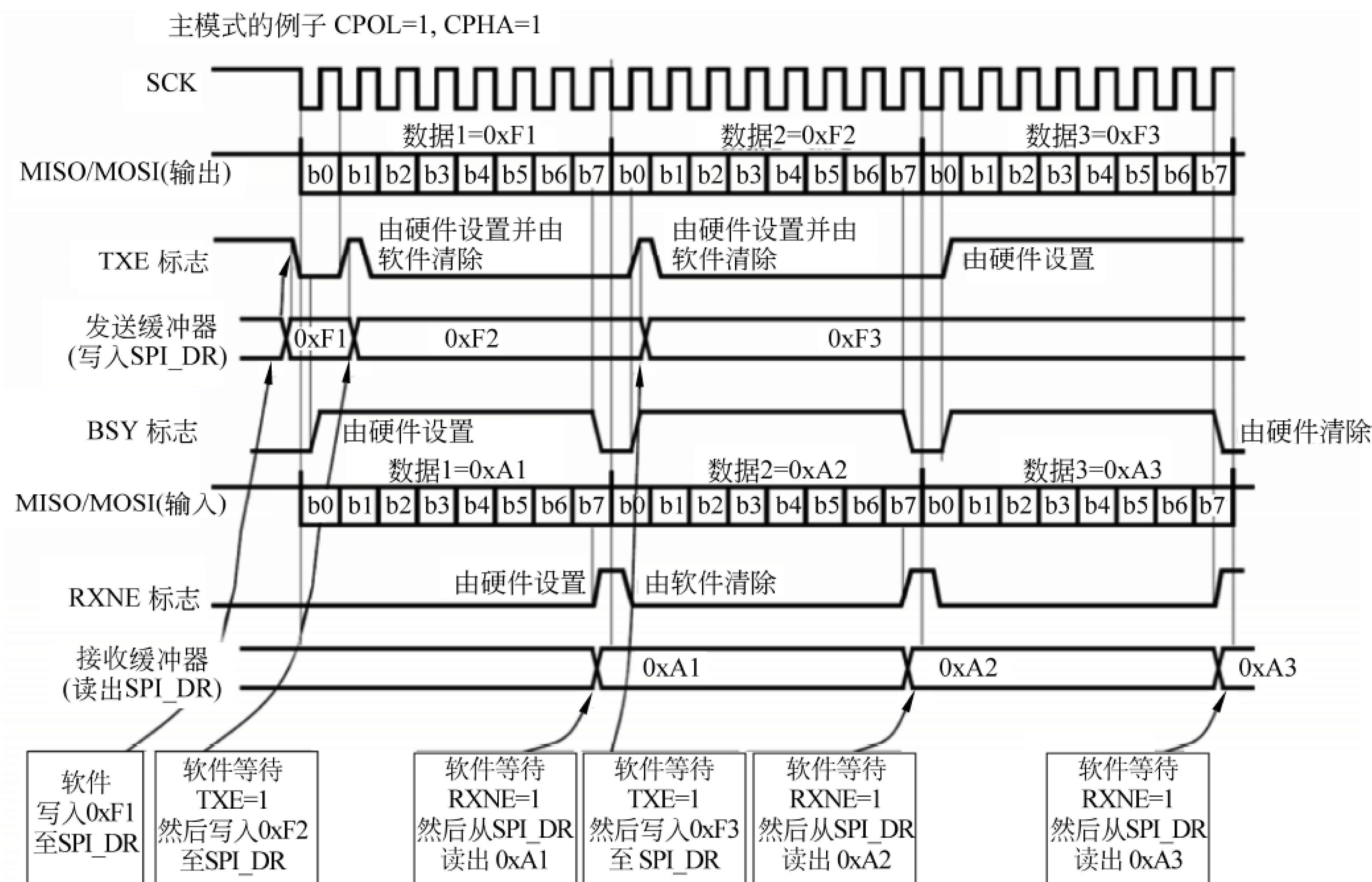


图 10-17　全双工主模式数据发送

全双工模式下发送和接收的处理流程如下：

- 设置 SPE 位为 1，使能 SPI 模块。
- 在 SPI_DR 寄存器中写入第一个要发送的数据。
- 等待 TXE＝1，然后写入第二个要发送的数据。
- 等待 RXNE＝1，然后读出 SPI_DR 寄存器并获得第一个接收到的数据，读 SPI_DR 的同时清除了 RXNE 位。重复步骤 3、4，发送后续的数据同时接收 $n-1$ 个数据。
- 等待 RXNE＝1，然后接收最后一个数据。
- 等待 TXE＝1，在 BSY＝0 之后关闭 SPI 模块。
- 如果开启了发送和接收中断，可以利用中断服务程序读写数据。

10.4.2　SPI 单工/半双工通信

在有些系统中，为节省 I/O 数量或者某些设备操作中主要以单向传输为主（如液晶屏），可以采用三线 SPI 接法。三线 SPI 只能实现单工/半双工通信，具体分为两种模式：

- 单线双向模式：1 条时钟线和 1 条双向数据线。
- 单线单向模式：1 条时钟线和 1 条单向数据线，只接收或只发送。

1. 单线双向模式

设置 SPI_CR1 寄存器中的 BIDIMODE 位为 1 启用单线双向模式。在这个模式下，SCK 引脚作为时钟，主设备使用 MOSI 引脚，从设备使用 MISO 引脚作为数据通信，即主设备 MOSI 和从设备 MISO 直接连接。传输的方向由 SPI_CR1 寄存器里的 BIDIOE 控制。当该位置 1 时数据线是输出（主设备发送数据），否则是输入（主设备接收数据）。BIDIMODE=1 并且 BIDIOE=1 时为双向发送，主设备数据线 MOSI 为输出，BIDIMODE=1 并且 BIDIOE=0 时为双向接收，主设备数据线 MOSI 为输入。

2. 单线单向模式

设置 SPI_CR1 寄存器中的 BIDIMODE 位为 0，且主设备 MOSI 连接到从设备 MISO 引脚时启用单线单向模式，在这个模式下，SPI 模块可以或者作为只发送，或者作为只接收，接收和发送状态由 RXONLY 位决定。

单向只发送模式（BIDIMODE=0 并且 RXONLY=0）：只发送模式和全双工模式类似，数据在发送引脚（主模式时是 MOSI、从模式时是 MISO）上传输，而接收引脚（主模式时是 MISO、从模式时是 MOSI）不使用，可以作为通用的 I/O 使用。程序中忽略接收缓冲器中的数据。

单向只接收模式（BIDIMODE=0 并且 RXONLY=1）：通过设置 SPI_CR2 寄存器的 RXONLY 位为 1 进入单向只接收模式，SPI 的输出功能被关闭，此时发送引脚（主模式时是 MOSI、从模式时是 MISO）可以作为 I/O 使用。在主设备中，一旦使能 SPI，即进入接收状态，BSY 标志始终置 1，关闭 SPI 时停止接收；在从设备中，一旦设备被片选（NSS=0）且 SCK 有时钟脉冲，SPI 处于接收状态。

全双工模式是双线单向模式，和单工的单线单向模式的区别在于，在单线单向模式下，主设备 MOSI 和从设备 MISO 连接，双线单向模式下，主从设备的 MOSI、MISO 分别和 MOSI、MISI 连接。

3. 单线模式数据发送与接收流程

单线只发送过程使用 BSY 位等待传输的结束，如果数据连续发送，从模式下的时序流程如图 10-18 所示。

- 设置 SPE 位为 1，使能 SPI 模块。
- 在 SPI_DR 寄存器中写入第一个要发送的数据。
- 等待 TXE=1，然后写入第二个要发送的数据；重复步骤 3，发送后续的数据。
- 写入最后一个数据到 SPI_DR 寄存器之后，等待 TXE=1；然后等待 BSY=0，完成数据传输。

主模式下的单线只发送和从模式下的区别在于在连续发送时 BSY 一直为高电平，不连续发送时与从模式单线只发送时序相同。

单向只接收模式的传输过程如图 10-19 所示，BSY 信号不起作用：

- 在 SPI_CR2 寄存器中，设置 RXONLY=1。

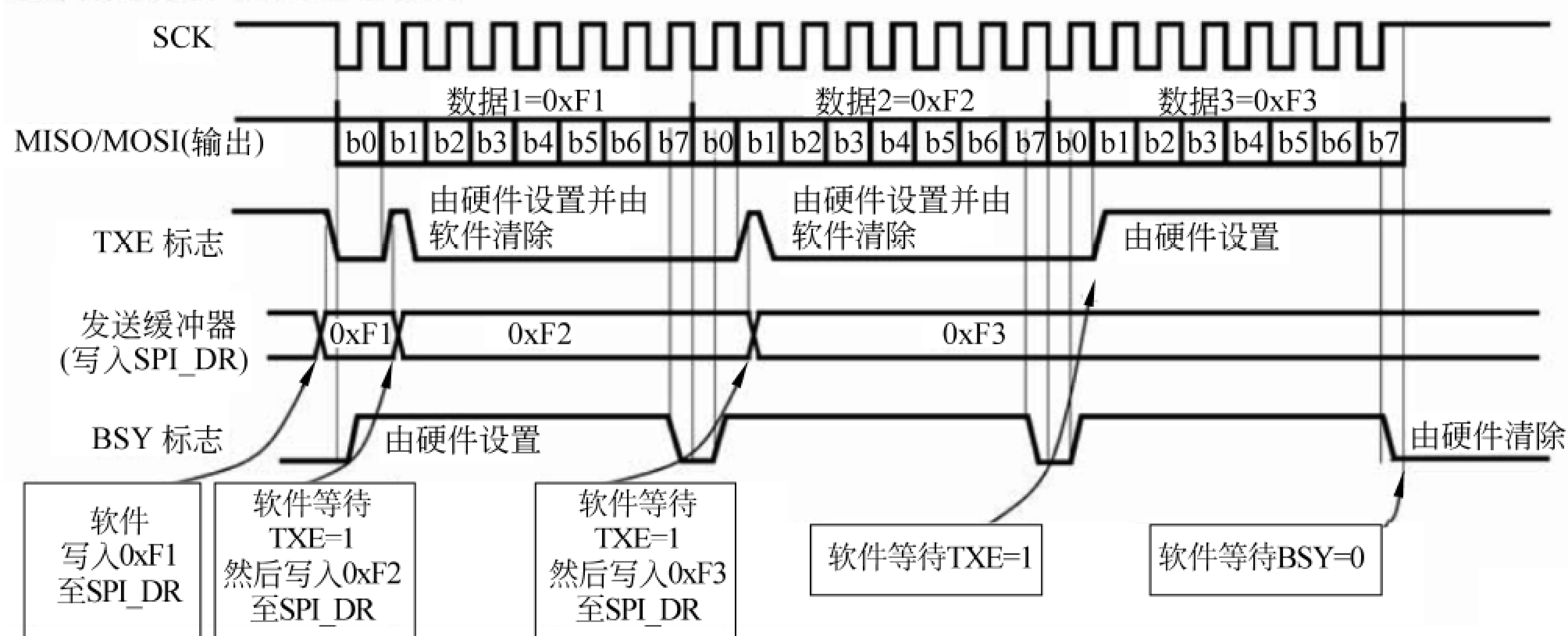


图 10-18 从模式单线只发送

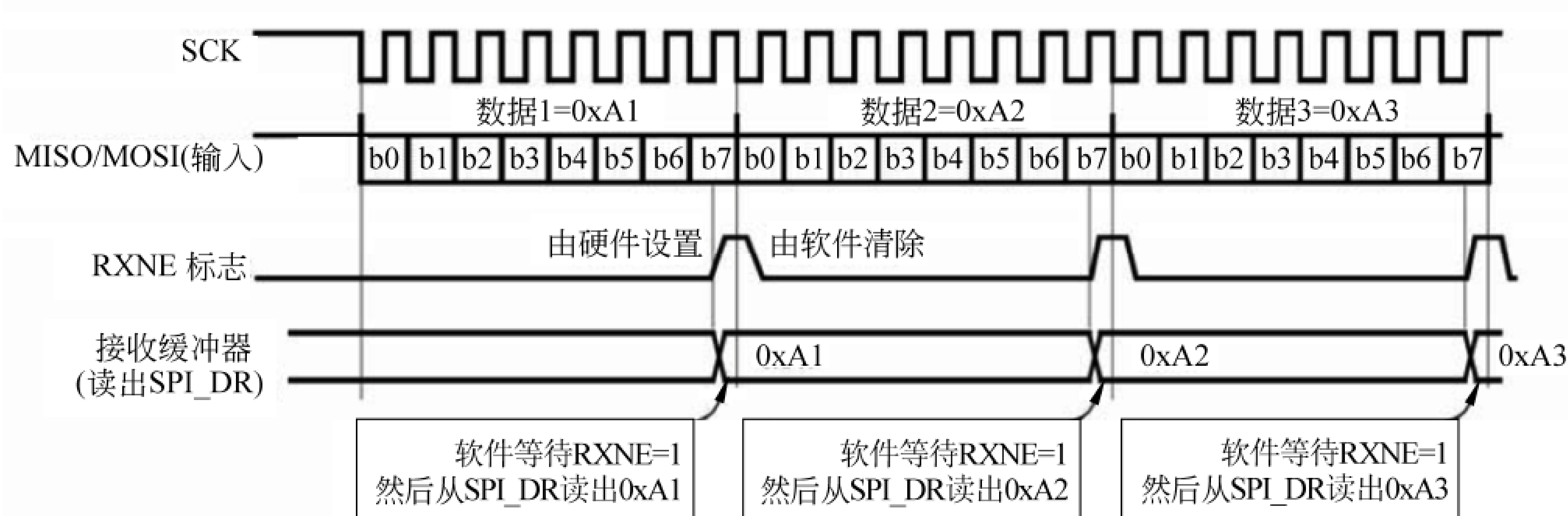


图 10-19 单线只接收模式

- 设置 SPE=1，使能 SPI 模块：
  - 主模式下立刻产生 SCK 信号，在关闭 SPI(SPE=0)之前不断地接收串行数据；
  - 从模式下，当 SPI 主设备拉低 NSS 信号并产生 SCK 时钟时，接收串行数据。
- 等待 RXNE=1，然后读出 SPI_DR 寄存器以获得收到的数据(同时会清除 RXNE 位)。重复此步骤接收所有数据。

单线双向传输的流程其发送过程和单线单向发送类似，接收过程和单线单向接收类似。

SPI 专用 GPIO 引脚如表 10-4 所示。

表 10-4 SPI GPIO 引脚

SPI 引脚功能	SPI1 I/O 引脚	SPI2 I/O 引脚	SPI3 I/O 引脚
SPI_NSS	PA4、PA15、PE12	PB12、PD0	PA15
SPI_SCK	PA5、PB3、PE13	PB13、PD1	PB3、PC10

续表

SPI 引脚功能	SPI1 I/O 引脚	SPI2 I/O 引脚	SPI3 I/O 引脚
SPI_MISO	PA6、PA11、PB4、PE14	PB14、PD3	PB4、PC11
SPI_MOSI	PA7、PA12、PB5、PE15	PB15、PD4	PB5、PC12

10.5 函数库

SPI 寄存器结构描述了固件函数库所使用的数据结构，固件库函数介绍了 ST 提供的典型库函数。

10.5.1 SPI 寄存器结构

SPI 寄存器结构，SPI_TypeDeff 在文件 stm32l1xx.h 中定义如下：

```
typedef struct
{
  __IO uint16_t CR1;                    // SPI 控制寄存器 1
  uint16_t RESERVED0;
  __IO uint16_t CR2;                    // SPI 控制寄存器 2
  uint16_t RESERVED1;
  __IO uint16_t SR;                     // SPI 状态寄存器
  uint16_t RESERVED2;
  __IO uint16_t DR;                     // SPI 数据寄存器
  uint16_t RESERVED3;
  __IO uint16_t CRCPR;                  // SPI CRC 多项式寄存器
  uint16_t RESERVED4;
  __IO uint16_t RXCRCR;                 // SPI 接收 CRC 寄存器
  uint16_t RESERVED5;
  __IO uint16_t TXCRCR;                 // SPI 发送 CRC 寄存器
  uint16_t RESERVED6;
} SPI_TypeDef;
```

2 个 SPI 外设声明文件 stm32l1xx.h：

```
#define PERIPH_BASE ((uint32_t)0x40000000)
#define APB1PERIPH_BASE PERIPH_BASE
#define APB2PERIPH_BASE (PERIPH_BASE + 0x10000)
#define AHBPERIPH_BASE (PERIPH_BASE + 0x20000)
#define SPI1_BASE (APB2PERIPH_BASE + 0x3000)
#define SPI2_BASE (APB1PERIPH_BASE + 0x3800)
```

```
#define SPI1 ((SPI_TypeDef * ) SPI1_BASE)
#define SPI2 ((SPI_TypeDef * ) SPI2_BASE)
```

SPI_InitTypeDef 定义于文件 stm32l1xx_spi.h,用于寄存器的初始化。

```
typedef struct
{
  uint16_t SPI_Direction;
  uint16_t SPI_Mode;
  uint16_t SPI_DataSize;
  uint16_t SPI_CPOL;
  uint16_t SPI_CPHA;
  uint16_t SPI_NSS;
  uint16_t SPI_BaudRatePrescaler;
  uint16_t SPI_FirstBit;
  uint16_t SPI_CRCPolynomial;
} SPI_InitTypeDef;
```

其中,SPI_Direction 设置了 SPI 单向或者双向的数据模式,取值为:

- SPI_Direction_2Lines_FullDuplex　SPI 设置为双线双向全双工
- SPI_Direction_2Lines_RxOnly　SPI 设置为双线单向接收
- SPI_Direction_1Line_Rx　SPI 设置为单线双向接收
- SPI_Direction_1Line_Tx　SPI 设置为单线双向发送

SPI_Mode 设置了 SPI 工作模式,取值为:

- SPI_Mode_Master　设置为主 SPI
- SPI_Mode_Slave　设置为从 SPI

SPI_DataSize 设置了 SPI 的数据大小,取值为:

- SPI_DataSize_16b　SPI 发送接收 16 位帧结构
- SPI_DataSize_8b　SPI 发送接收 8 位帧结构

SPI_CPOL 选择了串行时钟的默认值,取值为:

- SPI_CPOL_High　时钟悬空高
- SPI_CPOL_Low　时钟悬空低

SPI_CPHA 设置了位捕获的时钟活动沿的位置,取值为:

- SPI_CPHA_2Edge　数据捕获于第二个时钟沿
- SPI_CPHA_1Edge　数据捕获于第一个时钟沿

SPI_NSS 指定了 NSS 信号由硬件(NSS 引脚)还是软件(使用 SSI 位)管理,取值为:

- SPI_NSS_Hard　NSS 由外部引脚管理
- SPI_NSS_Soft　内部 NSS 信号有 SSI 位控制

SPI_BaudRatePrescaler 用来定义波特率预分频的值,这个值用以设置发送和接收的 SCK 时钟,取值为 SPI_BaudRatePrescalerX,其中 X 取值范围为 2、4、8、16、32、64、128、256,分别表示波特率预分频值为 X。

SPI_FirstBit 指定了数据传输从 MSB 位还是 LSB 位开始，取值为：

- SPI_FisrtBit_MSB　　　　数据传输从 MSB 位开始
- SPI_FisrtBit_LSB　　　　数据传输从 LSB 位开始

SPI_CRCPolynomial 定义了用于 CRC 值计算的多项式，默认值为 7。

10.5.2　SPI 库函数

SPI 库函数列表如表 10-5 所示。

表 10-5　SPI 库函数列表

函　数　名	描　　述
SPI_I2S_DeInit	将外设 SPIx 寄存器重设为默认值
SPI_Init	根据 SPI_InitStruc 中指定的参数初始化外设 SPIx 寄存器
SPI_StructInit	把 SPI_InitStruct 中的参数按默认值填入
SPI_DataSizeConfig	配置 8 位/16 位数据传输
SPI_Cmd	使能或者失能 SPI 外设
SPI_I2S_ITConfig	使能或者失能指定的 SPI 中断
SPI_DMACmd	使能或者失能指定 SPI 的 DMA 请求
SPI_I2S_SendData	通过外设 SPIx 发送一个数据
SPI_I2S_ReceiveData	返回通过 SPIx 最近接收的数据
SPI_NSSInternalSoftwareConfig	为选定的 SPI 软件配置内部 NSS 引脚
SPI_SSOutputCmd	使能或者失能指定的 SPI SS 输出
SPI_BiDirectionalLineConfig	选择指定 SPI 在双向模式下的数据传输方向
SPI_I2S_GetFlagStatus	检查指定的 SPI 标志位设置与否
SPI_I2S_ClearFlag	清除 SPIx 的待处理标志位
SPI_I2S_GetITStatus	检查指定的 SPI 中断发生与否
SPI_I2S_ClearITPendingBit	清除 SPIx 的中断待处理位
SPI_CalculateCRC	计算 CRC
SPI_TransmitCRC	传输 CRC
SPI_GetCRC	获取接收 CRC

在操作 SPI 控制器之前，需要打开 SPI 总线时钟，对 SPI1，调用 RCC_APB2PeriphClock-Cmd()，对 SPI2 调用 RCC_APB1PeriphClockCmd()。

1）函数 SPI_I2S_DeInit

功能描述：将外设 SPIx 寄存器重设为默认值。

函数原型：void SPI_I2S_DeInit(SPI_TypeDef * SPIx)。

输入参数 SPIx 用来选择 SPI 外设。

示例：

```
SPI_I2S_DeInit(SPI2);
```

2）函数 SPI_Init

功能描述：根据 SPI_InitStruct 中指定的参数初始化外设 SPIx 寄存器。

函数原型：void SPI_Init(SPI_TypeDef * SPIx, SPI_InitTypeDef * SPI_InitStruct)。

输入参数 SPIx 用来选择 SPI 外设,SPI_InitStruct 为指向结构 SPI_InitTypeDef 的指针,包含了外设 SPI 的配置信息。

示例：

```
SPI_InitTypeDef SPI_InitStructure;
SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex;
SPI_InitStructure.SPI_Mode = SPI_Mode_Master;
SPI_InitStructure.SPI_DatSize = SPI_DatSize_16b;
SPI_InitStructure.SPI_CPOL = SPI_CPOL_Low;
SPI_InitStructure.SPI_CPHA = SPI_CPHA_2Edge;
SPI_InitStructure.SPI_NSS = SPI_NSS_Soft;
SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_128;
SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB;
SPI_InitStructure.SPI_CRCPolynomial = 7;
SPI_Init(SPI1, &SPI_InitStructure);
```

3）函数 SPI_StructInit

功能描述：把 SPI_InitStruct 中的每一个参数按默认值填入。

函数原型：void SPI_StructInit(SPI_InitTypeDef * SPI_InitStruct)。

输入参数：SPI_InitStruct：指向结构 SPI_InitTypeDef 的指针,待初始化。默认值为：

```
SPI_Direction              SPI_Direction_2Lines_FullDuplex
SPI_Mode                   SPI_Mode_Slave
SPI_DataSize               SPI_DataSize_8b
SPI_CPOL                   SPI_CPOL_Low
SPI_CPHA                   SPI_CPHA_1Edge
SPI_NSS                    SPI_NSS_Hard
SPI_BaudRatePrescaler      SPI_BaudRatePrescaler_2
SPI_FirstBit               SPI_FirstBit_MSB
SPI_CRCPolynomial
```

示例：

```
SPI_InitTypeDef SPI_InitStructure;
SPI_StructInit(&SPI_InitStructure);
```

4）函数 SPI_Cmd

功能描述：使能或者失能 SPI 外设。

函数原型：void SPI_Cmd(SPI_TypeDef * SPIx, FunctionalState NewState)。

输入参数 SPIx 用于选择 SPI 外设，NewState 用于设置外设 SPIx 的状态，取值为 ENABLE 或者 DISABLE。

示例：

```
SPI_Cmd(SPI1, ENABLE);
```

5）函数 SPI_I2S_ITConfig

功能描述：使能或者失能指定的 SPI 中断。

函数原型：void SPI_I2S_ITConfig(SPI_TypeDef * SPIx, uint16_t SPI_I2S_IT, FunctionalState NewState)。

输入参数 SPIx 用来选择 SPI 外设，SPI_I2S_IT 为待使能或者失能的 SPI 中断源，其取值为：

- SPI_IT_TXE　　发送缓存空中断屏蔽
- SPI_IT_RXNE　　接收缓存非空中断屏蔽
- SPI_IT_ERR　　错误中断屏蔽

输入参数 NewState 为 SPIx 中断的状态，取值为 ENABLE 或者 DISABLE。

示例：

```
SPI_I2S_ITConfig(SPI2, SPI_IT_TXE, ENABLE);
```

6）函数 SPI_I2S_SendData

功能描述：通过外设 SPIx 发送一个数据。

函数原型：void SPI_I2S_SendData(SPI_TypeDef * SPIx, uint16_t Data)。

输入参数 SPIx 用来选择 SPI 外设，Data 为待发送的数据。

示例：

```
SPI_I2S_SendData(SPI1, 0xA5);
```

7）函数 SPI_I2S_ReceiveData

功能描述：返回通过 SPIx 最近接收的数据。

函数原型：uint16_t SPI_I2S_ReceiveData(SPI_TypeDef * SPIx)。

输入参数 SPIx 用来选择 SPI 外设。

示例：

```
uint16_t ReceivedData;
ReceivedData = SPI_I2S_ReceiveData(SPI2);
```

8）函数 SPI_NSSInternalSoftwareConfig

功能描述：为选定的 SPI 进行软件内部 NSS 引脚配置。

函数原型：void SPI_NSSInternalSoftwareConfig(SPI_TypeDef * SPIx, uint16_t SPI_NSSInternalSoft)。

输入参数 SPIx 用于选择 SPI 外设，SPI_NSSInternalSoft 为 SPI NSS 内部状态，其取值范围为：

- SPI_NSSInternalSoft_Set　　内部设置 NSS 引脚高电平
- SPI_NSSInternalSoft_Reset　　内部重置 NSS 引脚低电平

示例：

```
SPI_NSSInternalSoftwareConfig(SPI1, SPI_NSSInternalSoft_Set);
SPI_NSSInternalSoftwareConfig(SPI2, SPI_NSSInternalSoft_Reset);
```

9）函数 SPI_SSOutputCmd

功能描述：使能或者失能指定的 SPI NSS 引脚输出。

函数原型：void SPI_SSOutputCmd(SPI_TypeDef * SPIx, FunctionalState NewState)。

输入参数 SPIx 用来选择 SPI 外设，NewState 为 SPI NSS 引脚输出的状态，取值为 ENABLE 或者 DISABLE。

示例：

```
SPI_SSOutputCmd(SPI1, ENABLE);
```

10）函数 SPI_BiDirectionalLineConfig

功能描述：选择指定 SPI 在双向模式下的数据传输方向。

函数原型：SPI_BiDirectionalLineConfig(SPI_TypeDef * SPIx, uint16_t SPI_Direction)。

输入参数：SPIx 用来选择 SPI 外设，SPI_Direction 选择 SPI 在双向模式下的数据传输方向，取值为：

- SPI_Direction_Tx　　选择 Tx 发送方向
- SPI_Direction_Rx　　选择 Rx 接受方向

示例：

```
SPI_BiDirectionalLineConfig(SPI_Direction_Tx);
```

11）函数 SPI_I2S_GetFlagStatus

功能描述：检查指定的 SPI 标志位设置与否。

函数原型：FlagStatus SPI_GetFlagStatus(SPI_TypeDef * SPIx, uint16_t SPI_I2S_FLAG)。

输入参数 SPIx 用于选择 SPI 外设，SPI_I2S_FLAG 为待检查的 SPI 标志位，包括：

- SPI_FLAG_BSY　　忙标志位
- SPI_FLAG_OVR　　超出标志位

- SPI_FLAG_MODF　　　　模式错位标志位
- SPI_FLAG_CRCERR　　　　CRC 错误标志位
- SPI_FLAG_TXE　　　　发送缓存空标志位
- SPI_FLAG_RXNE　　　　接受缓存非空标志位

返回值为 SPI_I2S_FLAG 的状态，值为 SET 或者 RESET。

12）函数 SPI_I2S_ClearFlag

功能描述：清除 SPIx 的待处理标志位。

函数原型：void SPI_I2S_ClearFlag(SPI_TypeDef * SPIx, uint16_t SPI_I2S_FLAG)。

输入参数 SPIx 用于选择 SPI 外设，SPI_I2S_FLAG 为待清除的 SPI 标志位，取值与 SPI_I2S_GetFlagStatus 的 SPI_I2S_FLAG 相同，但是，不能清除标志位 BSY，TXE 和 RXNE，这三个标志位由硬件清零。

示例：

```
SPI_I2S_ClearFlag(SPI2, SPI_FLAG_OVR);
```

13）函数 SPI_I2S_GetITStatus

功能描述：检查指定的 SPI 中断发生与否。

函数原型：ITStatus SPI_I2S_GetITStatus(SPI_TypeDef * SPIx, uint8_t SPI_I2S_IT)。

输入参数 SPIx 用来选择 SPI 外设，SPI_I2S_IT 指定待检查的 SPI 中断源，包括：

- SPI_IT_OVR　　　　超出中断标志位
- SPI_IT_MODF　　　　模式错误标志位
- SPI_IT_CRCERR　　　　CRC 错误标志位
- SPI_IT_TXE　　　　发送缓存空中断标志位
- SPI_IT_RXNE　　　　接受缓存非空中断标志位

返回值为 SPI_I2S_IT 的新状态，值为 SET 或 RESET。

示例：

```
ITStatus Status;
Status = SPI_I2S_GetITStatus(SPI1, SPI_IT_OVR);
```

14）函数 SPI_I2S_ClearITPendingBit

功能描述：清除 SPIx 的中断待处理位。

函数原型：void SPI_I2S_ClearITPendingBit(SPI_TypeDef * SPIx, uint8_t SPI_I2S_IT)。

输入参数 SPIx 用于选择 SPI 外设，SPI_I2S_IT 为待清除的 SPI 中断源，与 SPI_I2S_Get_ITStatus 函数的第二个参数取值相同，中断标志位 BSY，TXE 和 RXNE 由硬件重置。

示例：

```
SPI_I2S_ClearITPendingBit(SPI2, SPI_IT_CRCERR);
```

10.6 SPI 案例

10.6.1 SPI 寄存器操作案例

1）SPI_Iint 函数实现

```
void SPI_Init(SPI_TypeDef * SPIx, SPI_InitTypeDef * SPI_InitStruct)
{
  uint16_t tmpreg = 0;
  //CR1寄存器配置
  tmpreg = SPIx->CR1;
  //清除 BIDIMode, BIDIOE, RxONLY, SSM, SSI, LSBFirst, BR, MSTR, CPOL 和 CPHA
  tmpreg &= CR1_CLEAR_MASK;
  //根据 SPI_Direction 配置 BIDImode, BIDIOE 和 RxONLY 字段
  //根据 SPI_Mode 和 SPI_NSS 设置 SSM, SSI 和 MSTR 字段
  //根据 SPI_FirstBit 设置 LSBFirst
  //根据 SPI_BaudRatePrescaler 设置 BR 字段
  //根据 SPI_CPOL 和 SPI_CPHA 设置 CPOL 和 CPHA 字段
  tmpreg |= (uint16_t)((uint32_t)SPI_InitStruct->SPI_Direction
          |SPI_InitStruct->SPI_Mode | SPI_InitStruct->SPI_DataSize
          |SPI_InitStruct->SPI_CPOL | SPI_InitStruct->SPI_CPHA
          |SPI_InitStruct->SPI_NSS  | SPI_InitStruct->SPI_BaudRatePrescaler
          |SPI_InitStruct->SPI_FirstBit);
  SPIx->CR1 = tmpreg;
  //配置 CRC多项式寄存器
  SPIx->CRCPR = SPI_InitStruct->SPI_CRCPolynomial;
}
```

2）中断状态读取函数实现

```
ITStatus SPI_I2S_GetITStatus(SPI_TypeDef * SPIx, uint8_t SPI_I2S_IT)
{
  ITStatus bitstatus = RESET;
  uint16_t itpos = 0, itmask = 0, enablestatus = 0;
  //根据 SPI_I2S_IT 的定义获取状态寄存器对应的中断状态字段的位置
  //例如 TXE中断定义为 0x71,7表示中断控制 TXEIE的位置,1表示 SR寄存器 TXE标志的位置
  itpos = 0x01 << (SPI_I2S_IT & 0x0F);
  //根据输入的中断源计算中断开关的字段位置
  itmask = SPI_I2S_IT >> 4;
  itmask = 0x01 << itmask;
  //判断中断源所对应的中断开关字段是否被打开
```

```
    enablestatus = (SPIx->CR2 & itmask) ;
    //如果允许中断且状态寄存器对应的状态位为 1,则返回 SET,否则返回 RESET
    if (((SPIx->SR & itpos) != (uint16_t)RESET) && enablestatus)
      bitstatus = SET;
    else
      bitstatus = RESET;
    return bitstatus;
}
```

10.6.2 SPI 函数库案例

1）基本配置流程

- 调用 RCC_APB2PeriphClockCmd(RCC_APB2Periph_SPI1，ENABLE)或 RCC_APB1Periph ClockCmd(RCC_APB1Periph_SPI2，ENABLE)使能 SPI1 和 SPI2 的时钟。
- 调用 RCC_AHBPeriphClockCmd()使能 SCK，MOSI，MISO，NSS 引脚对应的 GPIO 端口时钟。
- 调用 GPIO_PinAFConfig()将 I/O 与复选功能进行映射。
- 配置 GPIO 引脚为复选功能，上拉或下拉输出，调用 GPIO_Init 进行初始化。
- 配置极性、相位、波特率分频，主从模式等，通过 SPI_Init()进行初始化。
- 配置 NVIC，使能相应中断，调用 SPI_ITConfig()开启 SPI 中断源。
- 调用 SPI_Cmd()启用 SPI。
- 也可以通过调用 SPI_BiDirectionalLineConfig()配置三线模式，SPI_NSSInternalSoftwareConfig()配置 NSS 引脚，SPI_DataSizeConfig()配置数据位宽以及调用 SPI_SSOutputCmd()设置 NSS 的输入输出状态。

2）库函数配置案例

```
void SPI_INIT()
{
    SPI_InitTypeDef SPI_InitStructure;
    GPIO_InitTypeDef GPIO_InitStructure;
    SPI_Cmd(SPI2,DISABLE);
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOB, ENABLE);               //使能 GPIOB 时钟
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_SPI2, ENABLE);
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_15;                        // 初始化指定引脚
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_40MHz;
    GPIO_Init(GPIOB, &GPIO_InitStructure);
```

```
    GPIO_PinAFConfig(GPIOB, GPIO_PinSource15, GPIO_AF_SPI2);
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_14;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_40MHz;
    GPIO_Init(GPIOB, &GPIO_InitStructure);
    GPIO_PinAFConfig(GPIOB, GPIO_PinSource14, GPIO_AF_SPI2);
    SPI_InitStructure.SPI_Direction= SPI_Direction_2Lines_FullDuplex;//全双工
    SPI_InitStructure.SPI_Mode= SPI_Mode_Master;          //SPI 主设备
    SPI_InitStructure.SPI_DataSize= SPI_DataSize_8b;      //SPI 传输数据 8 位
    SPI_InitStructure.SPI_CPOL= SPI_CPOL_Low;             //时钟极性为低跳变到高采集数据
    SPI_InitStructure.SPI_CPHA= SPI_CPHA_1Edge;           //时钟相位(第一个跳变沿采集数据)
    SPI_InitStructure.SPI_NSS= SPI_NSS_Soft;              //软件片选方式
    SPI_InitStructure.SPI_BaudRatePrescaler= SPI_BaudRatePrescaler_16;//1 分频
    SPI_InitStructure.SPI_FirstBit= SPI_FirstBit_MSB;     //数据传输从 MSB 位开始
    SPI_InitStructure.SPI_CRCPolynomial= 7;               //设置 crc 多项式
    SPI_Init(SPI2,&SPI_InitStructure);
    SPI_SSOutputCmd(SPI2, ENABLE);                        //主模式,NSS 引脚输出
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12;
    GPIO_InitStructure.GPIO_Mode= GPIO_OType_PP;
    GPIO_Init(GPIOB, &GPIO_InitStructure);
    SPI_Cmd(SPI2,ENABLE);
}
数据发送
while(1)
{
    while(SPI_I2S_GetFlagStatus(SPI2,SPI_I2S_FLAG_BSY )!=RESET);
    SPI_I2S_SendData(SPI2, 0x01);
    while(SPI_I2S_GetFlagStatus(SPI2,SPI_I2S_FLAG_BSY )!=RESET);
}
```

10.6.3　温度传感器 ADT7320 案例

和 I2C 一样,对于特定的外设,其数据的读写有特定的字节传输时序要求,ADT7320 是一款 SPI 接口的温度传感器,其结构如图 10-20 所示。其中,SCLK、DOUT、DIN 和 CS 为 SPI 总线接口,DIN 为 MOSI,DOUT 为 MISO。

ADT7320 内部有多个寄存器,用于存储温度值和对传感器芯片进行配置,因此在对传感器操作时,我们需要通过 SPI 总线向 ADT7320 发送一个命令字,命令字中定义了对寄存器的读写操作方向和所要访问的寄存器地址,在主设备向 ADT7320 发送命令字时,主设备收到的数据无效,命令字格式如图 10-21 所示。

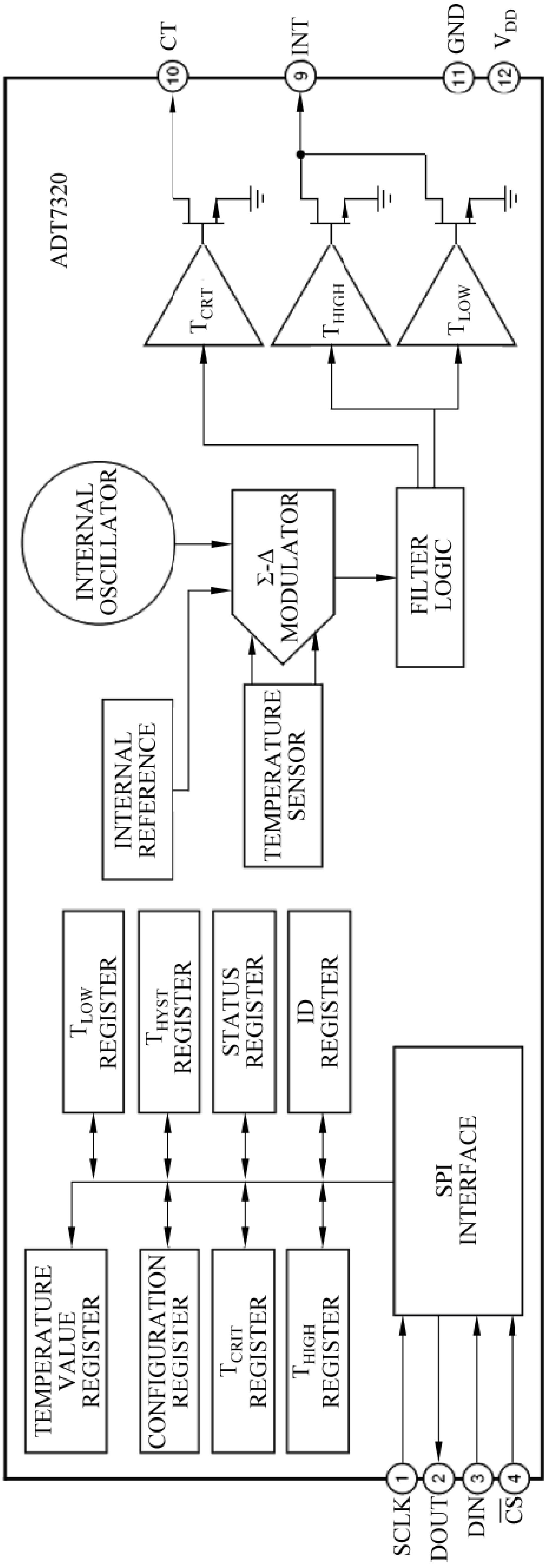


图 10-20 ADT7320 传感器芯片结构

C7	C6	C5	C4	C3	C2	C1	C0
0	R/$\overline{W}$	寄存器地址			0	0	0

图 10-21　命令字格式

ADT7320 向寄存器写一个字节的数据时序如图 10-22 所示，主设备首先发送命令字，命令字中 C6 置为 0，C5～C3 给出所要写的寄存器地址，然后紧接着主设备再发送一个字节的数据，从设备 ADT7320 发回的数据忽略。

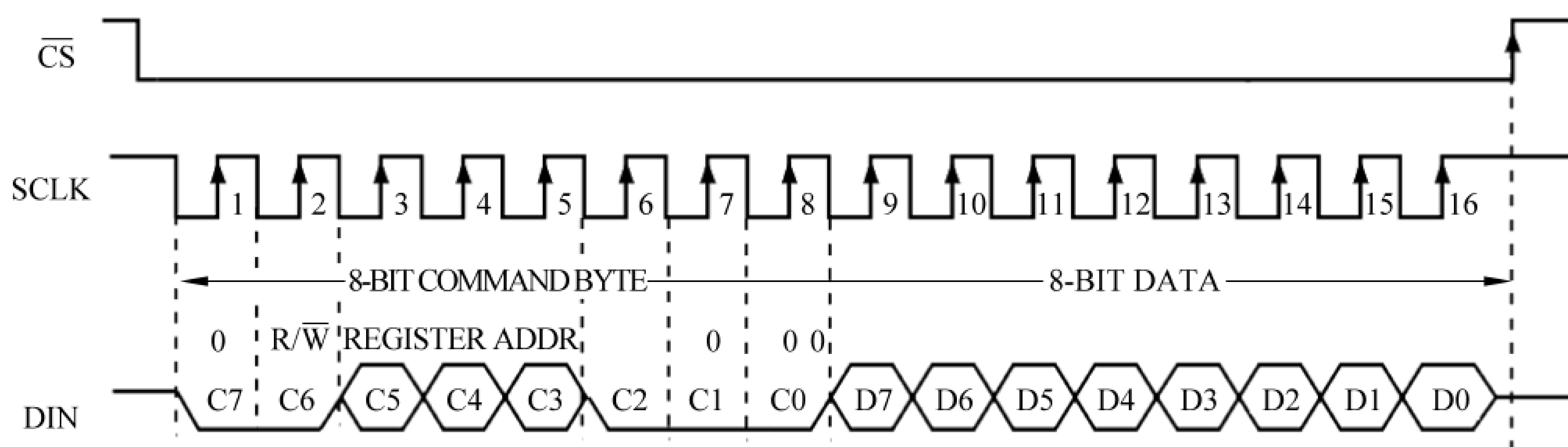


图 10-22　SPI 写寄存器时序

读寄存器值时，首先需要向 ADT7320 发送一个命令字，C6 置为 0，从机 ADT7320 的返回数据无效，然后主设备再向从设备发送一个任意数据，从设备将之前收到的命令字中寄存器地址所对应的数据发送到主设备，完成一次寄存器读操作，时序如图 10-23 所示。

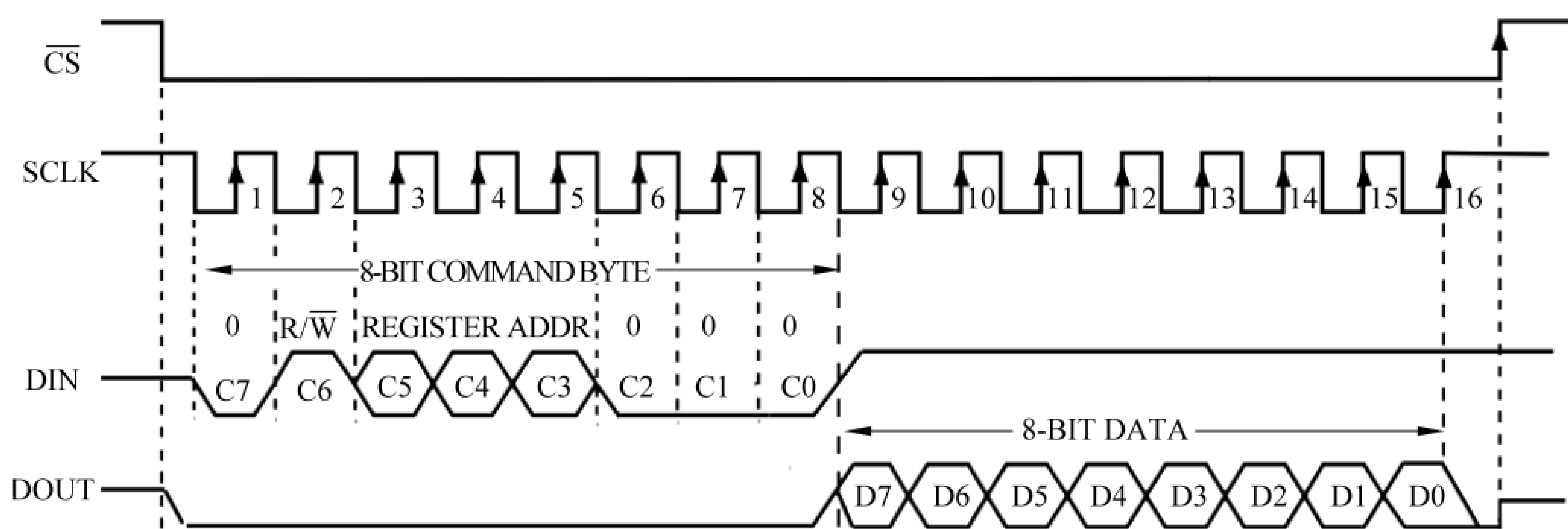


图 10-23　读寄存器时序

一个 I/O 模拟的读温度数值的函数实现如下：

```
unsigned int ReadFromADT7320ViaSPI(unsigned int reg_address)
{
    unsigned int   i;
    unsigned int spi_misoValue;
    unsigned int spi_Value;
```

```
    spi_Value = ( 0x78 & ((reg_address + 8) << 3));      //命令字
    OutputBit(CS,1);
    OutputBit(SCL,1);
    OutputBit(CS,0);
    Delay(5);
    for(i=0;i<8;i++)                                     //发送命令字
    {
        OutputBit(SCL,0);
        if((spi_Value & 0x80)==0x80)
            OutputBit(DIN,1);
        else
            OutputBit(DIN,0);
        Delay(5);
        OutputBit(SCL,1);
        spi_Value = (spi_Value <<1);
        Delay(5);
    }
    //读取寄存器的值
    for(i=0;i<8;i++)
    {
        OutputBit(SCL,0);
        spi_misoValue = (spi_misoValue <<1);
        Delay(10);
        if(InputBit(DOUT) ==1)
            spi_misoValue |=0x0001;
        else
            spi_misoValue &=0xfffe;
        Delay(2);
        OutputBit(SCL,1);
        Delay(8);
    }
    OutputBit(SCL,1);
    OutputBit(CS,1);
    return  spi_misoValue;
}
```

Chapter 11

第 11 章　模拟/数字转换

【导读】 在数据采集系统中,模数转换是至关重要的环节,在精度要求不高的情况下通常采用嵌入式微控制集成的模拟数字器实现模数转换。本章首先介绍模拟数字器采集的相关概念,然后对 STM32L152 片内集成 AD 的结构、寄存器和库函数使用方法进行了介绍,并对模拟数字器基本初始化和使用进行了案例阐述。

11.1　ADC 简介

1. 模拟信号和数字信号

模拟信号指信息参数在给定范围内表现为连续的信号,或在一段连续的时间间隔内,其代表信息的特征量可以在任意瞬间呈现为任意数值的信号,例如电压、电流与声音等。

数字信号指幅度的取值是离散的,幅值表示被限制在有限个数值之内。二进制码就是一种数字信号。二进制码受噪声的影响小,易于有数字电路进行处理,所以得到了广泛的应用。

2. 模拟数字转换器

模拟数字转换器(Analog-to-digital converter, ADC)是用于将模拟形式的连续信号转换为数字形式的离散信号的器件。典型的模拟数字转换器将模拟信号转换为表示一定比例电压值的数字信号。

通常的模数转换器是把经过与标准量比较处理后的模拟量转换成以二进制数值表示的离散信号的转换器,其工作原理如图 11-1 所示。输入的模拟信号 V(t)在采样时钟 CP 的控制下将连续量转变成了离散量,两个离散的采样点之间的信号由取样保持电路负责保持前

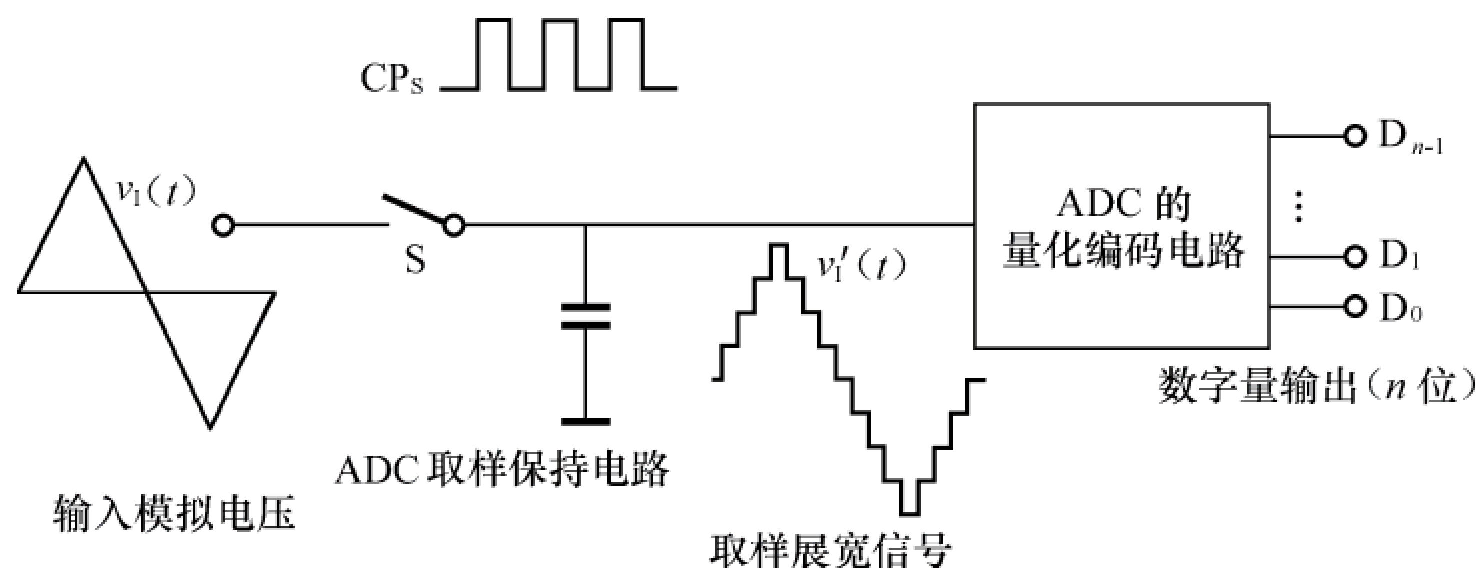


图 11-1　ADC 采样原理

一个采样点的电压。这样将“光滑”的模拟信号变成了“齿状”的取样展宽信号，对该信号和参考信号进行比较，衡量其大小，即可输出二进制编码，类似于十进制到二进制的转换。故任何一个模数转换器都需要一个参考模拟量作为转换的标准，比较常见的参考标准为最大的可转换信号大小，而输出的数字量则表示输入信号相对于参考信号的大小。

3. AD 采样过程

A/D 转换过程是通过取样、保持、量化和编码这四个步骤完成的采样样和保持主要由采样保持器来完成，量化编码由 A/D 转换器完成。

1）采样

采样是对模拟信号进行周期性抽样取值的过程，采样将连续的模拟信号分解成许多个离散点。如图 11-2 所示，采样开关在采样时钟的控制下对输入信号进行采样，为了保证采样后的信号能恢复模拟信号的特征，采样的时钟必须满足奈奎斯特采样定律，即采样频率应不小于输入模拟信号频谱中最高频率的两倍：Fsample >= 2 * Fmax_signal。

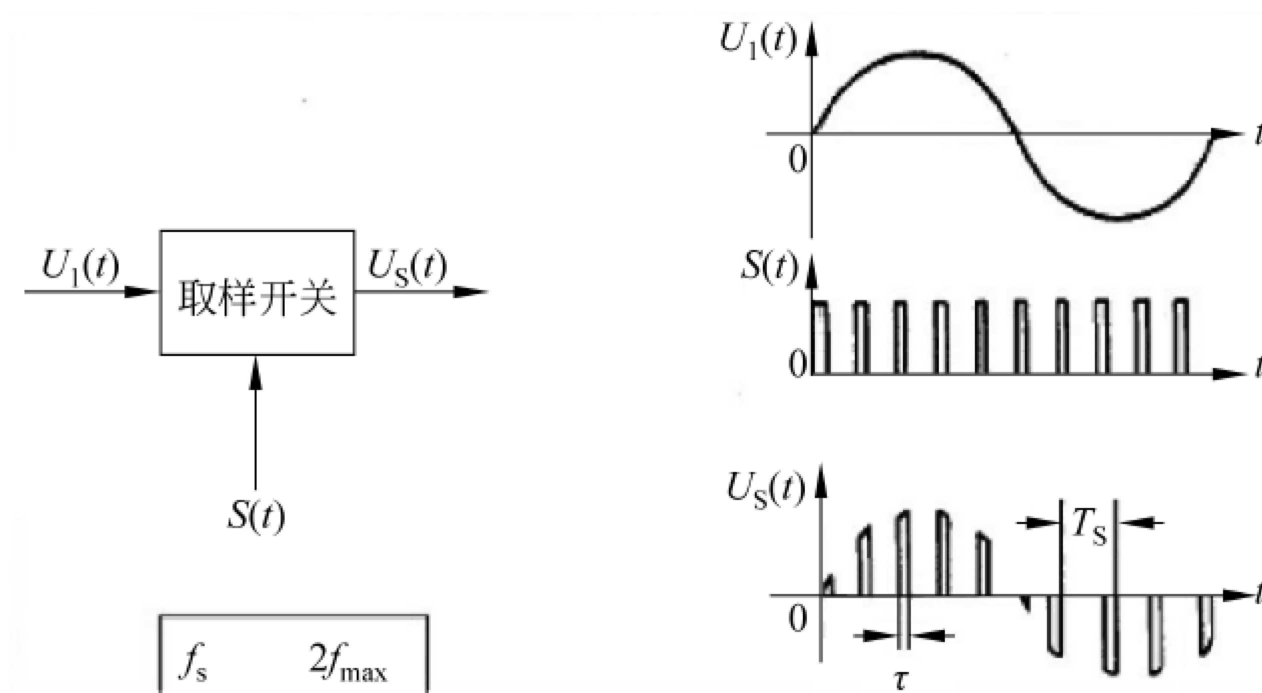


图 11-2 采样过程

2）保持

保持，即将采样点的电压信号保持一段时间。因为后续的量化过程需要一定的时间 τ，对于随时间变化的模拟输入信号，要求瞬时采样值在时间 τ 内保持不变，这样才能保证转换的正确性和转换精度，这个过程就是保持。如图 11-3 所示，采样保持后，原来连续模拟信号变成了阶梯形的连续信号。

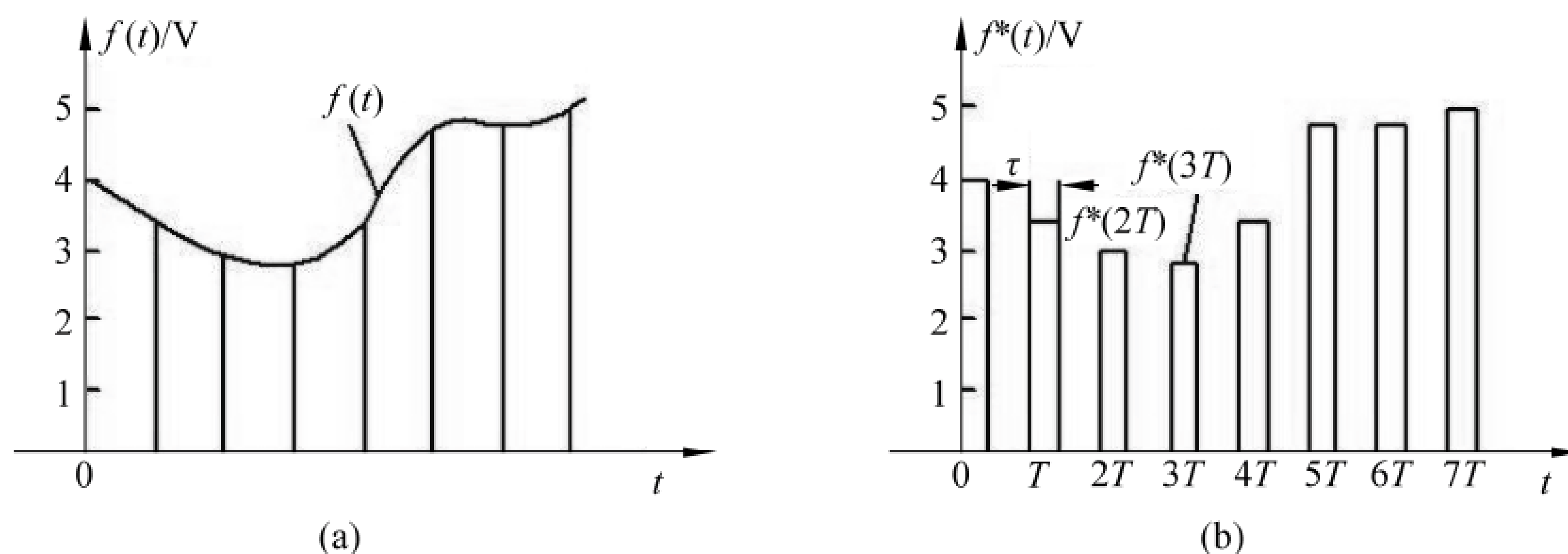


图 11-3 采样保持过程

3）量化

量化是对输入信号的幅值按照量化单位进行离散处理的过程。即采样将时间轴进行了离散化，量化将采样信号的幅值经过舍入或截尾的方法进行离散化，变为只有有限个有效数字的数。若取信号可能出现的最大值 A，令其分为 D 个间隔，则每个间隔长度为 R=A/D，R 称为量化增量或量化步长，当采样信号落在某一小间隔内，经过舍入或截尾方法而变为有限值时，则产生量化误差，ADC 的量化位数一般采用 8，12，16，24 位，量化位数越大，分辨率越高，量化误差越小。

4）编码

编码是将一系列模拟信号采样和量化后用数字信号给描述出来的过程，这个数字信号序列就是编码。

4. ADC 的性能指标

衡量一个 ADC 的性能指标包括：

1）分辨率

分辨率（Resolution）是指 ADC 转化器所能分辨的模拟信号的最小变化值，分辨率的高低取决于量化位数，n 位的量化下，数字量变化一个最小量时模拟信号的变化量为：Vmax $*1/2^n$。例如 8 位的 AD，可以描述 256 个刻度的精度（2 的 8 次方），在它测量一个 0～5V 电压信号时，它的分辨率是 5V 除以 256，0.02V，即最小单位 0.02V。

2）转换速率

转换速率（Conversion Rate）是完成一次从模拟转换到数字的 A/D 转换所需的时间，转换速率决定了采样频率的最大值，因此间接影响了输入模拟信号的信号频率的最大值。不同类型的 ADC 转换速率不同，积分型 ADC 一次转化在毫秒级、逐次比较型 ADC 一次转换在微秒级，并行比较型 ADC 可以达到纳秒级。

3）量程

量程是 A/D 转换的输入信号电压范围，一般为 0～5V、0～10V、－5～＋5V、－10～＋10V，可以根据具体的输入信号进行调整。

4）信号连接方式

A/D 转换的输入信号可以采用单端方式或差分方式。单端方式的信号共用一个模拟地，抗干扰能力较差，但能提供更多的输入通道；查分方式每个通道需要两个引脚，信号之间互不影响，可对差分信号进行采集，抗干扰能力强。

5）量化误差

量化误差（Quadratuer Error）是指由于对模拟信号进行量化而产生的误差，量化结果和被量化模拟量的差值，显然量化位数越多，量化的相对误差越小，A/D 的量化误差为 1 或 1/2 的最小量化步长（取决于量化时的舍入方式是截断还是四舍五入），如图 11-4 所示，最差情况下是一个量化步长，最好情况是 1/2 量化步长。

6）偏移误差

偏移误差（Offset Error）是指输入信号为零时输出信号不为零引起的误差值，一般以满量程电压值的百分比表示。大多数的偏移误差是由温度引起的，可以通过外部电路进行调零。

7）满刻度误差

满刻度误差（Full Scale Error）也叫增益误差，满度输出时对应的输入信号与理想输入

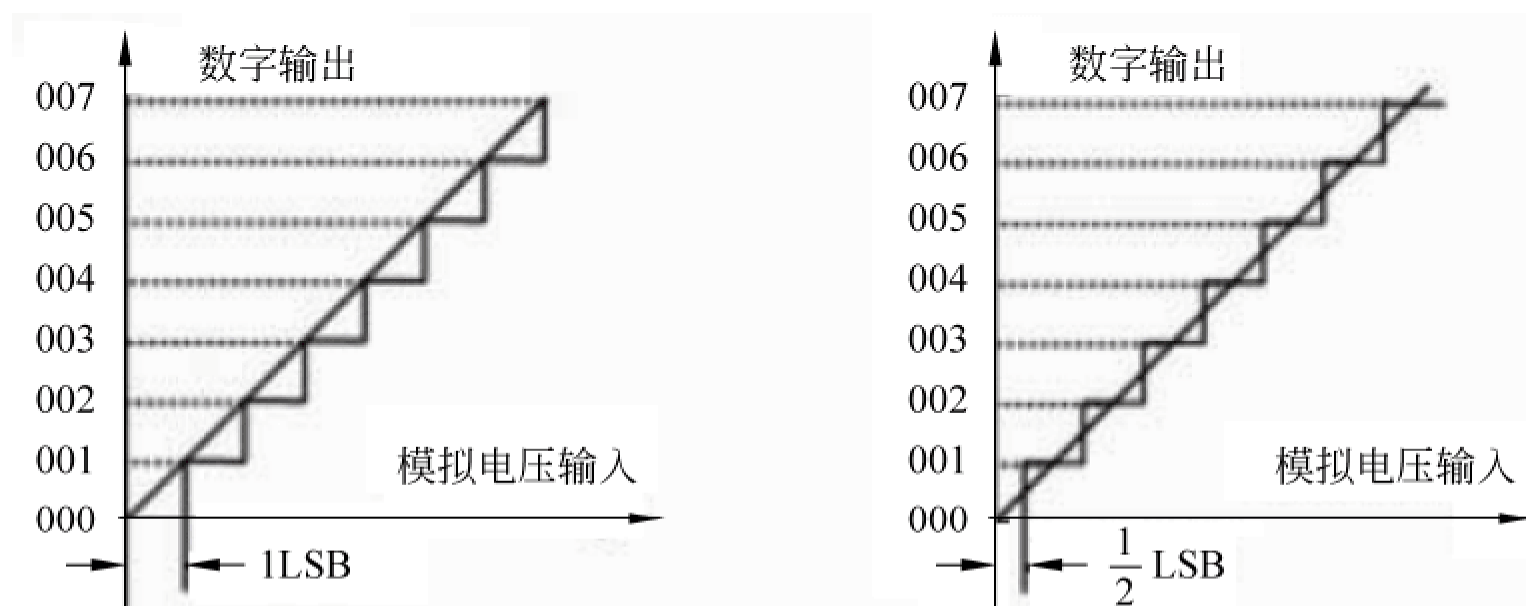


图 11-4 量化误差

信号值之差,也采用满量程电压值的百分比表示。

8) 线性度

线性度(Linearity)指在没有偏移和增益误差的情况下,实际转换器的转移函数与理想直线的最大偏移。线性误差是由 ADC 特性随输入信号幅度变化引起的,因此不能补偿。

5. A/D 转换器类型

A/D 转换器主要分为积分型、逐次逼近型、并行比较型/串并行型、Σ-Δ 调制型、电容阵列逐次比较型及压频变换型等。

1) 积分型

积分型 A/D 工作原理是将输入电压转换成时间(脉冲宽度信号)或频率(脉冲频率),然后由定时器/计数器获得数字值。其优点是用简单电路就能获得高分辨率,但缺点是由于转换精度依赖于积分时间,因此转换速率极低。初期的单片 A/D 转换器大多采用积分型,现在逐次比较型已逐步成为主流。

2) 逐次逼近型

逐次逼近型 A/D 由一个比较器和 D/A 转换器通过逐次比较逻辑构成,从 MSB 开始,顺序地对每一位将输入电压与内置 D/A 转换器输出进行比较,经 n 次比较而输出数字值。其电路规模属于中等。其优点是速度较高、功耗低,在低分辨率(<12 位)时价格便宜,但高精度(>12 位)时价格很高。

3) 并行比较型/串并行比较型

并行比较型 A/D 采用多个比较器,仅作一次比较而实行转换,又称 Flash(快速)型。由于转换速率极高,n 位的转换需要 $2n-1$ 个比较器,因此电路规模也极大,价格也高,只适用于视频 A/D 转换器等速度特别高的领域。

串并行比较型 A/D 结构上介于并行比较型和逐次逼近型之间,最典型的是由 2 个 $n/2$ 位的并行型 A/D 转换器配合 D/A 转换器组成,用两次比较实行转换,所以称为 Half flash(半快速)型。还有分成三步或多步实现 A/D 转换的叫做分级(Multistep/Subrangling)型 A/D,而从转换时序角度又可称为流水线型 A/D,现代的分级型 A/D 中还加入了对多次转换结果作数字运算而修正特性等功能。这类 A/D 速度比逐次比较型高,电路规模比并行型小。

4) Σ-Δ 调制型

Σ-Δ 型 A/D 由积分器、比较器、1 位 DA 转换器和数字滤波器等组成。原理上近似于积

分型，将输入电压转换成时间（脉冲宽度）信号，用数字滤波器处理后得到数字值。电路的数字部分基本上容易单片化，因此容易做到高分辨率。主要用于音频和测量。

5）电容阵列逐次比较型

电容阵列逐次比较型 A/D 在内置 D/A 转换器中采用电容矩阵方式，也可称为电荷再分配型。一般的电阻阵列 D/A 转换器中多数电阻的值必须一致，在单芯片上生成高精度的电阻并不容易。如果用电容阵列取代电阻阵列，可以用低廉成本制成高精度单片 A/D 转换器。最近的逐次比较型 A/D 转换器大多为电容阵列式的。

6）压频变换型

压频变换型（Voltage-Frequency Converter）是通过间接转换方式实现模数转换的。其原理是首先将输入的模拟信号转换成频率，然后用计数器将频率转换成数字量。从理论上讲这种 A/D 的分辨率几乎可以无限增加，只要采样的时间能够满足输出频率分辨率要求的累积脉冲个数的宽度。其优点是分辨率高、功耗低、价格低，但是需要外部计数电路共同完成 A/D 转换。

6. 逐次逼近型 ADC 的工作原理

图 11-5 为逐次逼近型的结构图。这种 A/D 转换器是以 D/A 转换器为基础，加上比较器、逐次逼近寄存器、置数选择逻辑电路及时钟等组成。

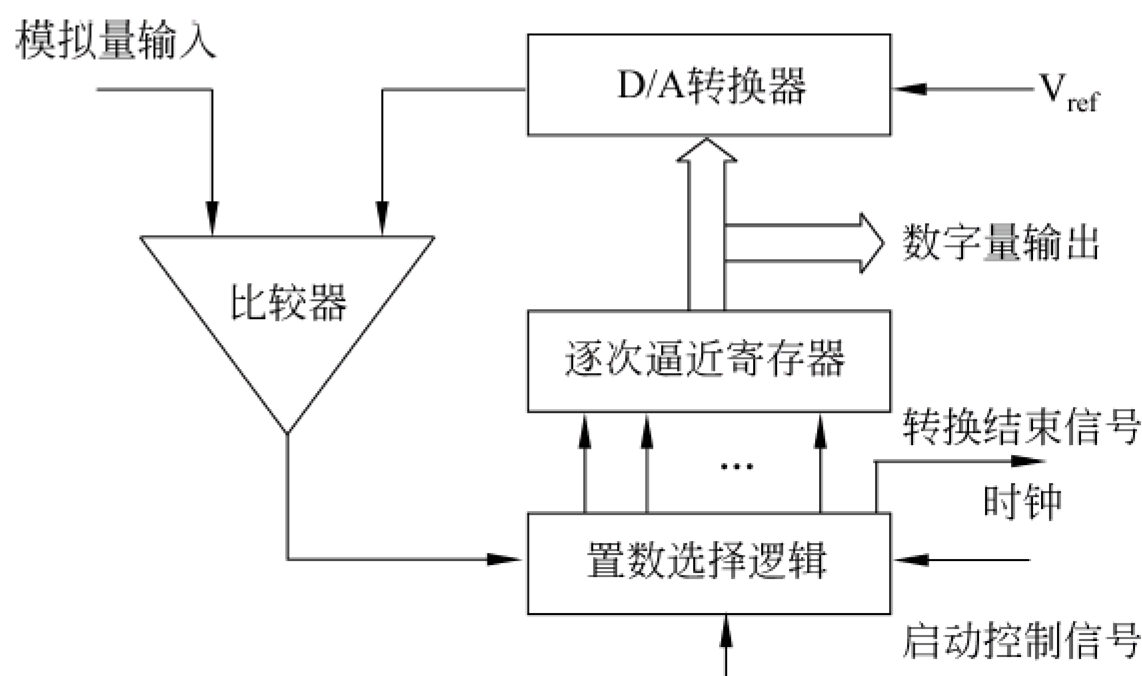


图 11-5　逐次逼近型电路结构图

其转换原理如图 11-6 所示。在启动信号控制下，首先置数选择逻辑电路，给逐次逼近寄存器最高位置 1，经 D/A 转换成模拟量后与输入模拟量进行比较，电压比较器给出比较结果。如果输入量大于或等于经 D/A 变换后输出的量，则比较器为 1，否则为 0，置数选择逻辑电路根据比较器输出的结果，修改逐次逼近寄存器中的内容，使其经 D/A 变换后的模拟量逐次逼近输入模拟量。这样经过若干次修改后的数字量，便是 A/D 转换结果的量。

逼近型 A/D 大多采用二分搜索法，即首先取允许电压最大范围的 1/2 值与输入电压值进行比较，也就是首先最高为 1，其余位为 0。如果搜索值在此范围内，则再取范围的 1/2 值，即次高位置 1。如果搜索值不在此范围内，则应以搜索值的最大允许输入电压值的另外 1/2 范围，即最高位为 0，依次进行下去，每次比较将搜索范围缩小 1/2，具有 n 位的 A/D 变换，经 n 次比较，即可得到结果。因此，必须在 A/D 转换结束后才能从逐次逼近寄存器中取出数字量。为此 D/A 芯片专门设置了转换结束信号引脚，向 CPU 发转换结束信号，通知 CPU 读取转换

后的数字量，CPU 可以通过中断或查询方式检测 A/D 转换结束信号，并从 A/D 芯片的数据寄存器(即图 10-9 中逐次逼近寄存器)中取出数字量。逐次逼近法变换速度较快，所以集成化的 A/D 芯片多采用上述方法，STM32L152 内部集成的 ADC 便是逐次逼近型 ADC。

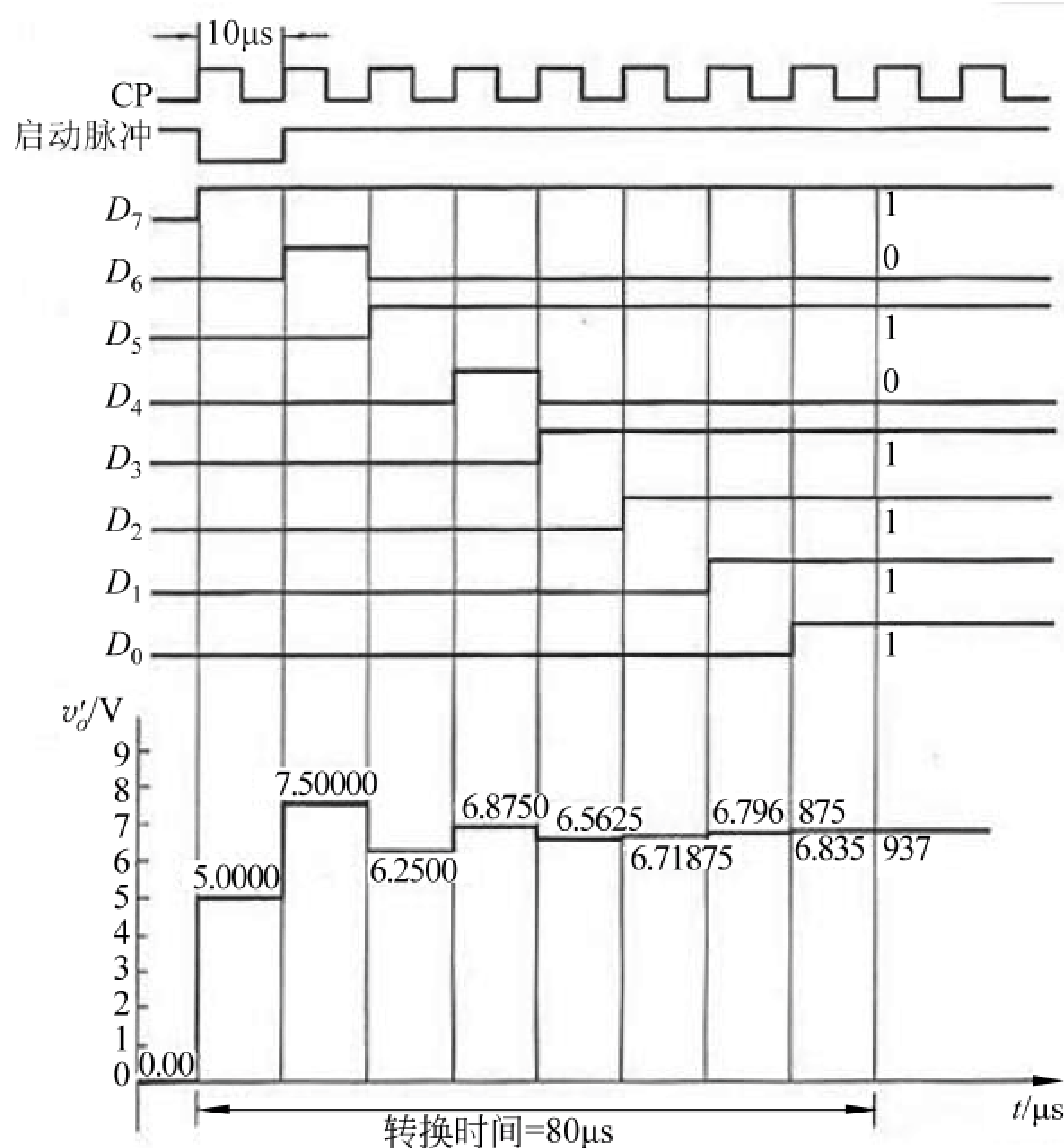


图 11-6　逐次逼近型 ADC 转换原理

11.2　STM32L152 ADC

STM32 内部集成了一个 12 位的逐次逼近型模拟数字转换器，最多支持 42 个通道，可以测量 40 个外部引脚的信号和 2 个内部信号，每个通道支持单次、连续、扫描或间断模式，转换结果以左对齐或右对齐方式存储在 16 位数据寄存器中。在给定的系统时钟驱动下，ADC 转换默认以最快速度执行，同时支持动态电源管理，旨在 ADC 转换期间供电，以降低功耗，其主要特征为：

- 12 位、10 位、8 位、6 位可配置的量化位数。
- 转换结束、注入转换结束和发生模拟看门狗事件、溢出事件时产生中断。
- 支持单次和连续转换模式。
- 支持可编程通道次序的自动扫描模式。
- 转换结果支持左对齐或右对齐方式。
- 每个 ADC 通道支持单独的采样时间配置。
- 规则通道和注入通道均有外部触发选项。

- ADC 转换时间：最大转换速率 1μs(ADCCLK=16MHz)到 4μs(ADCCLK=4MHz)。
- 可设置的自动关机模式以降低功耗。
- ADC 供电要求：高速 2.4～3.6V，低速 1.8V。
- ADC 输入范围：VREF−≤VIN≤VREF+。

其内部结构如图 11-7 所示。

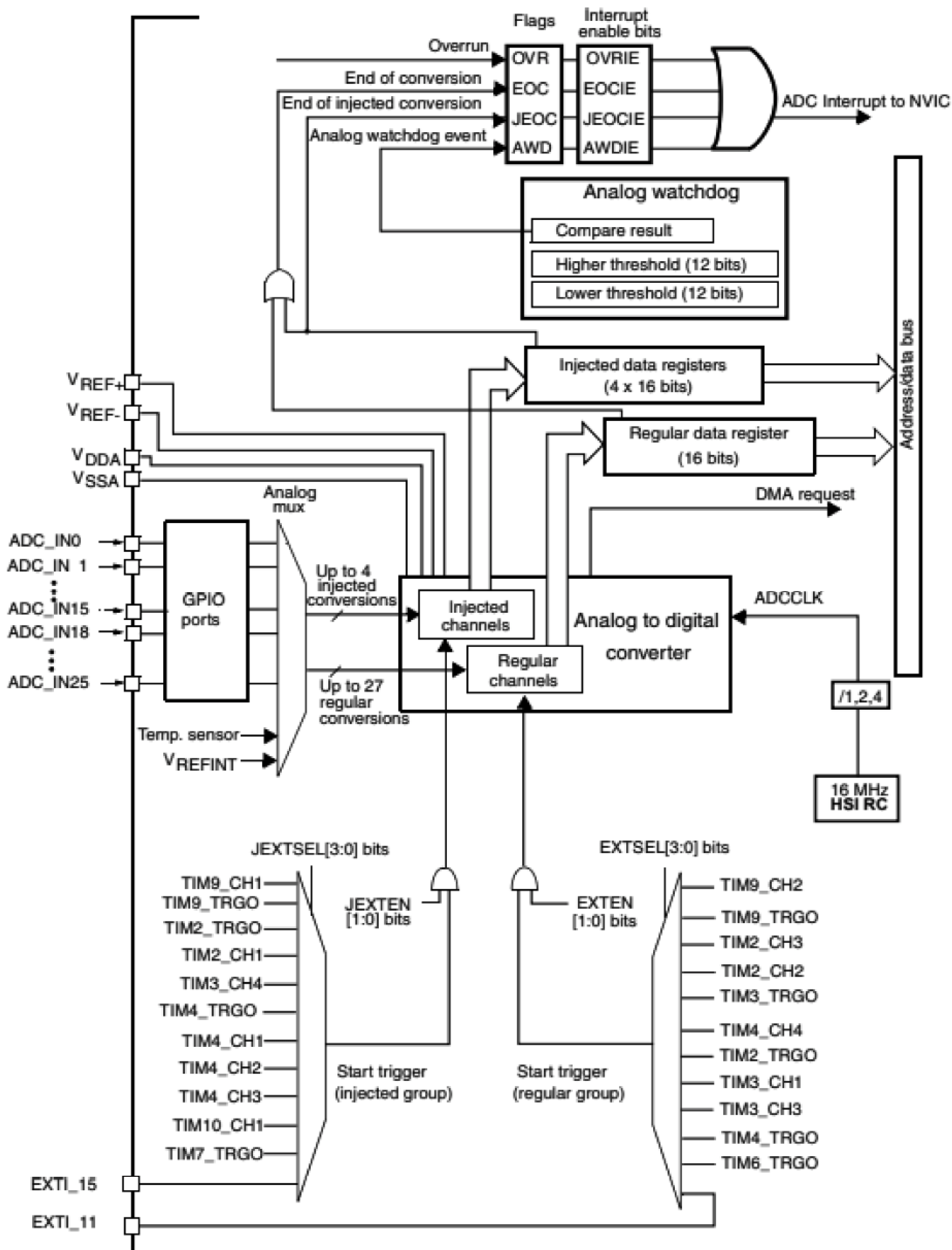


图 11-7　ADC 内部结构图

ADC 对外的接口见表 11-1 所示。一般情况下，VDD 小于 3.6V，VSS 接地，相对应的，VDDA 小于 3.6V，VSSA 也接地，模拟输入信号不要超过 VDD。

表 11-1　ADC 接口定义

名称	信 号 类 型	注　　解
VREF＋	输入，模拟参考正极	ADC 使用的参考电压，全速 ADCCLK＝16MHz，VREF＋ ＝VDDA≥2.4V；中速 ADCCLK＝8MHz 要求 VREF＋ ＝VDDA≥1.8V，若 VREF＋≠VDDA，则 VREF＋需大于 2.4V；低速 ADCCLK＝4MHz，VREF＋需大于 1.8V
VDDA	输入，模拟电源	等效于 VDD 的模拟供电电源， 全速要求 2.4V≤VDDA≤VDD(3.6V)， 中低速要求 1.8V≤VDDA≤VDD(3.6V)
VREF－	输入，模拟参考负极	ADC 使用的负极参考电压，VREF－＝VSSA
VSSA	输入，模拟电源地	等效于 VSS 的模拟电源地
ADC_INx	模拟输入信号	21～40 通道模拟输入通道

STM32L152 内部集成了 1 个 ADC，支持 21 个外部通道，可以作为 ADC 输入的 GPIO 引脚如表 11-2 所示。

表 11-2　STM32L152RET6 的 ADC 外部引脚

ADC_INx	GPIO	ADC_INx	GPIO
ADC_IN0	PA0	ADC_IN11	PC1
ADC_IN1	PA1	ADC_IN12	PC2
ADC_IN2	PA2	ADC_IN13	PC3
ADC_IN3	PA3	ADC_IN14	PC4
ADC_IN4	PA4	ADC_IN15	PC5
ADC_IN5	PA5	ADC_IN18	PB12
ADC_IN6	PA6	ADC_IN19	PB13
ADC_IN7	PA7	ADC_IN20	PB14
ADC_IN8	PB0	ADC_IN21	PB15
ADC_IN9	PB1	ADC_IN0b	PB2
ADC_IN10	PC0		

11.2.1　STM32L152 ADC 功能

STM32L52RET6 带 1 个 ADC 控制器，一共支持 23 个通道，包括 21 个外部和 2 个内

部信号源，内部信号源 ADC_IN16 和 ADC_IN17 分别被连接到了温度传感器和内部参照电压 VREFINT 上。

1. ADC 供电控制

通过设置 ADC_CR2 寄存器的 ADON 位可给 ADC 上电。当第一次设置 ADON 位时，它将 ADC 从断电状态下唤醒。通过清除 ADON 位可以停止转换，并将 ADC 置于断电模式。在这个模式中，ADC 几乎不耗电。ADC 上电后，上电延迟一段时间后（tSTAB）当 SWSTART、JSWSTART 位或者外部触发信号到达时，ADC 启动转换。

为降低功耗，当 ADC 转换就绪时（ADONS＝1），ADC 自动对电源进行管理，通过 ADC_CR1 寄存器的 PDI 和 PDD 位，在 ADC 没有转换时自动进入掉电状态。PDI 用于配置 ADC 在等待软件或外部信号触发转换的状态是否自动掉电，PDD 用于表示两个转换之间的延迟内 ADC 是否自动掉电。

2. ADC 时钟配置

从图 11-8 可以看出，ADC 的模拟部分时钟（即采样时钟）来源独立于总线时钟，采用内部高速时钟 HSI，HSI 最高 16MHz，即无论 MCU 主频多少，ADC 的最高速率为 16MHz。

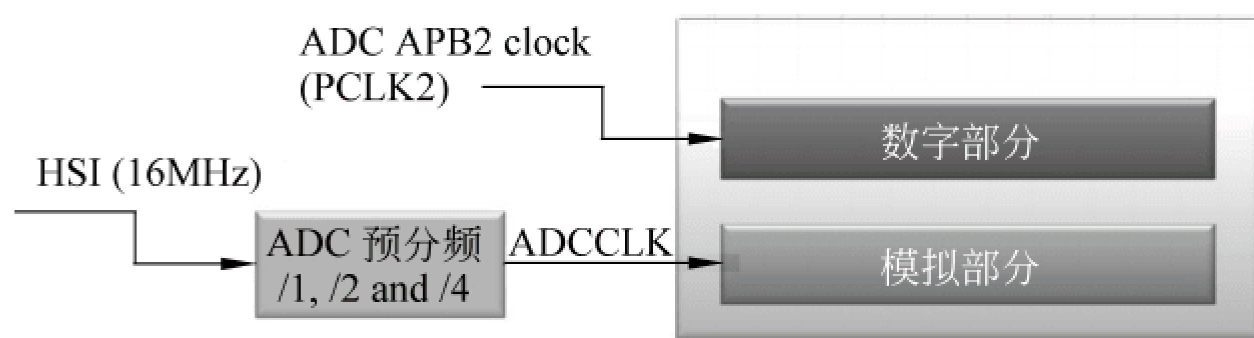


图 11-8　ADC 时钟

ADC 的工作时钟为 ADCCLK，ADCCLK 通过 ADC 预分频器可以对 HSI 时钟进行分频：

- 1 分频时为全速，ADCCLK＝16MHz；
- 2 分频时为中速，ADCCLK＝8MHz；
- 4 分频时为低速，ADCCLK＝4MHz；

ADC 控制器挂接在 APB2 总线上，数字部分的数据交互需要使用总线时钟，由于 ADC 采样时钟可能会高于总线时钟，此时有可能引起 ADC 采集的数据无法及时被 CPU 取走，可以通过注入延迟等方法降低数据采集的速度。ADC 的总线接口通常由时钟控制器提供的 ADCCLK 时钟和 PCLK2（APB2 时钟）同步。RCC 控制器为 ADC 时钟提供一个专用的可编程预分频器。

3. ADC 通道选择

STM32 的 ADC 控制器有很多通道，所以模块通过内部的模拟多路开关，可以切换到不同的输入通道并进行转换。在任意多个通道上以任意顺序进行的一系列转换构成成组转换。例如，可以如下顺序完成转换：通道 3、通道 8、通道 2、通道 2、通道 0、通道 2、通道 2、通道 15。STM32 特别地加入了多种成组转换的模式，可以由程序设置好之后，对多个模拟通道自动地进行逐个地采样转换。它们可以组织成两组：规则通道组和注入通道组。

(1) 规则组由多达 28 个转换组成。规则通道和它们的转换顺序在 ADC_SQRx 寄存器中选择。规则组中转换的总数应写入 ADC_SQR1 寄存器的 L[3：0]位中。

(2) 注入组由多达 4 个转换组成。注入通道和它们的转换顺序在 ADC_JSQR 寄存器中选择。注入组里的转换总数目应写入 ADC_JSQR 寄存器的 L[1：0]位中。

如果 ADC_SQRx 或 ADC_JSQR 寄存器在转换期间被更改，当前的转换被清除，一个新的启动脉冲将发送到 ADC 以转换新选择的组。

规则组和注入组的关系如图 11-9 所示。规则通道组的转换是按照既定的序列正常执行，而注入通道组的转换则是打断规则组的执行优先执行的一组转换，类似于程序和中断服务程序。

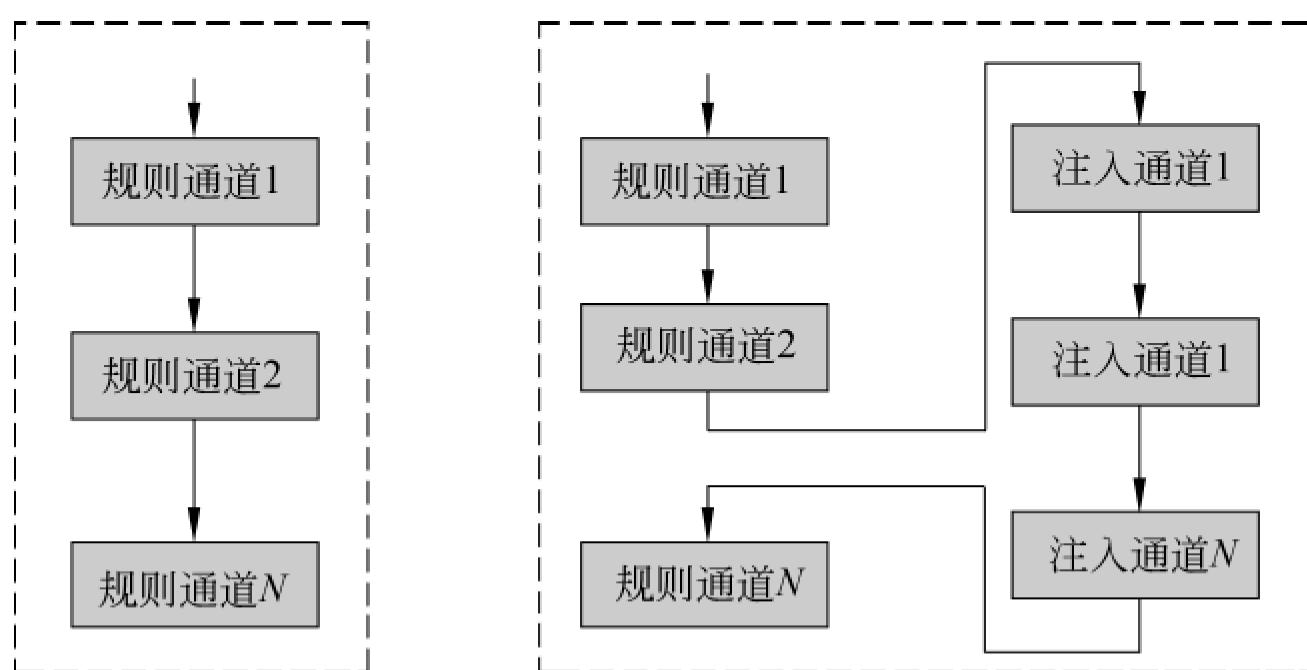


图 11-9　规则组和注入组的对比

4. 注入通道管理

注入通道可以通过软件或外部触发进行注入，也可以通过软件设置自动注入。

(1) 触发注入：使用触发注入时，必须清除 ADC_CR1 寄存器的 JAUTO 位，具体方式如下：

- 利用外部触发或通过设置 ADC_CR2 寄存器的 JSWSTART 位，启动一组注入通道的转换；
- 如果在规则通道转换期间产生一外部注入触发，当前转换被复位，注入通道序列被以单次扫描方式进行转换。
- 恢复上次被中断的规则组通道转换。

如果在注入转换期间产生一规则事件，注入转换不会被中断，但是规则序列将在注入序列结束后被执行。

(2) 自动注入：如果设置了 JAUTO 位，在规则组通道之后，注入组通道被自动转换。这可以用来转换在 ADC_SQRx 和 ADC_JSQR 寄存器中设置的最多 31 个转换序列。自动注入模式中，必须禁止注入通道的外部触发，如果除 JAUTO 位外还设置了 CONT 位，规则通道至注入通道的转换序列被连续执行。

5. ADC 转换方式

ADC 支持四种转换模式：单通道单次转换、单通道连续转换、多通道单次转换、多通道

连续转换，其关系如图 11-10 所示。



图 11-10　四种转换模式

1）单次转换模式

单次转换模式下，ADC 只执行一次转换。该模式下 ADC_CR2 寄存器的 CONT 为 0，当设置 ADC_CR2 寄存器的 SWSTART 位（规则通道）或 JSWSTART 位（注入通道）或者外部触发时（规则、注入通道均可）转换开始，转换完成后 ADC 停止。

- 如果一个规则通道被转换：

— 转换数据被储存在 16 位 ADC_DR 寄存器中；

— EOC（转换结束）标志被设置；

— 如果设置了 EOCIE，则产生中断。

- 如果一个注入通道被转换：

— 转换数据被储存在 16 位的 ADC_DRJ1 寄存器中；

— JEOC（注入转换结束）标志被设置；

— 如果设置了 JEOCIE 位，则产生中断。

2）连续转换模式

在连续转换模式中，当前面 ADC 转换一结束马上就启动另一次转换。此模式可通过外部触发启动或通过设置 ADC_CR2 寄存器上的 ADON 位启动，此时 CONT 位是 1。

- 如果一个规则通道被转换：

— 转换数据被储存在 16 位的 ADC_DR 寄存器中；

— EOC（转换结束）标志被设置；

— 如果设置了 EOCIE，则产生中断。

转换通道在 SQR5 寄存器的 SQ1[4：0]中指定。注入通道无法进行连续转换，只有注入通道被配置成规则通道之后的连续转换模式时（JAUTO=1）才能进行注入通道转换：

— 转换数据被储存在 16 位的 ADC_DRJ1 寄存器中

— JEOC（注入转换结束）标志被设置；

— 如果设置了 JEOCIE 位，则产生中断。

转换时序如图 11-11 所示。

3）扫描模式

扫描模式用于对一组模拟通道进行转换。扫描模式可通过设置 ADC_CR1 寄存器的

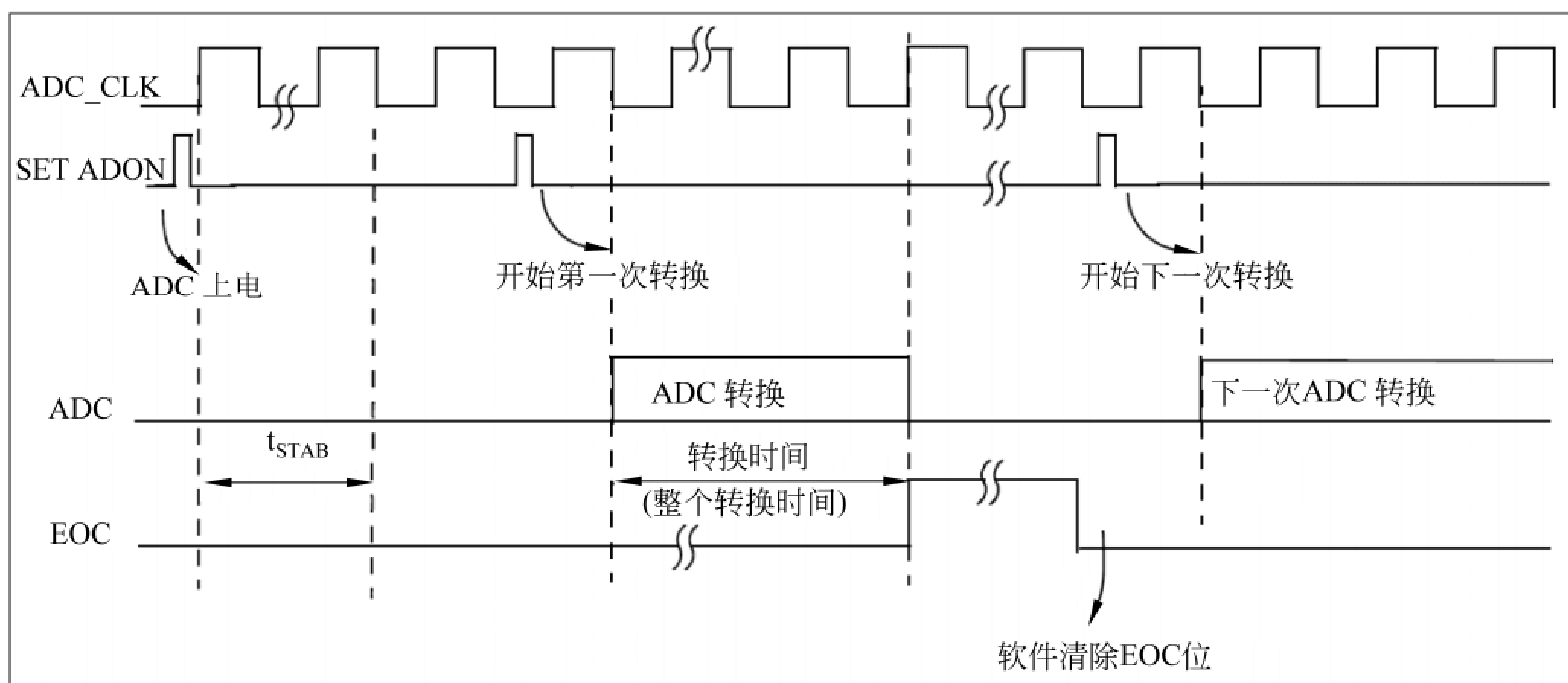


图 11-11　ADC 转换时序

SCAN 位来选择。一旦这个位被设置，ADC 扫描所有被 ADC_SQRX 寄存器(规则通道)或 ADC_JSQR(注入通道)选中的所有通道。在每个组的每个通道上执行单次转换。在每个转换结束时，同一组的下一个通道被自动转换。如果设置了 CONT＝1，则转换不会在选择组的最后一个通道上停止，而是再次从选择组的第一个通道继续转换，即多通道连续扫描模式，否则为多通道单次扫描模式。规则通道的每一组转换完成后状态寄存器 ADC_SR 的 EOC 清零，每个规则组的通道转换完后状态寄存器 ADC_SR 的 EOC 位置 1。如果在使用扫描模式的情况下使用中断，会在最后一个通道转换完毕后才会产生中断。而连续转换，是在每次转换后，都会产生中断。

4）间断模式

间断模式是一种特殊的扫描模式，对于规则组通道，此模式通过设置 ADC_CR1 寄存器上的 DISCEN 位激活。它可以用来执行一个子序列的 n 次转换($n \leqslant 8$)，此转换是 ADC_SQRx 寄存器所选择的转换序列的一部分。数值 n 由 ADC_CR1 寄存器的 DISCNUM[2：0]位给出。

一个外部触发信号可以启动 ADC_SQRx 寄存器中描述的下一轮 n 次转换，直到此序列所有的转换完成为止。所有的规则组的子序列转换完成后，下次开始从第一个子序列开始转换，总的序列长度由 ADC_SQR1 寄存器的 L[3：0]定义。例如：$n=3$，被转换的通道＝0、1、2、3、6、7、9、10。

- 第一次触发：转换的序列为 0、1、2；
- 第二次触发：转换的序列为 3、6、7；
- 第三次触发：转换的序列为 9、10，并产生 EOC 事件；
- 第四次触发：转换的序列 0、1、2。

对于注入组，此模式通过设置 ADC_CR1 寄存器的 JDISCEN 位激活。在一个外部触发事件后，该模式用于执行 ADC_JSQR 寄存器的一个子序列的 n 次转换($n \leqslant 3$)，n 由 ADC_

CR1 寄存器的 DISCNUM[2：0]位给出。

一个外部触发信号可以启动 ADC_JSQR 寄存器选择的下一个通道序列的转换，直到序列中所有的转换完成为止。总的序列长度由 ADC_JSQR 寄存器的 JL[1：0]位定义，例如 $n=1$，被转换的通道＝1、2、3。

- 第一次触发：通道 1 被转换；
- 第二次触发：通道 2 被转换；
- 第三次触发：通道 3 被转换，并且产生 EOC 和 JEOC 事件；
- 第四次触发：通道 1 被转换。

规则组转换的另一个例子如图 11-12 所示。ADC_SQR1 的值为：0、1、2、4、5、8、9、11、12、13、14、15；间隔转换的通道数量为 3。



图 11-12　规则通道转换案例

6. 数据对齐

ADC 转换的数据保存在数据寄存器，数据寄存器 16 位，但 ADC 最高支持到 12 位，因此数据的存储可以以左对齐或右对齐的方式，如图 11-13 和图 11-14 所示，ADC_CR2 寄存器中的 ALIGN 位选择转换后数据储存的对齐方式。

注入组

SEXT	SEXT	SEXT	SEXT	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

规则组

0	0	0	0	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

图 11-13　数据左对齐

注入组

SEXT	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	0	0	0

规则组

D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	0	0	0	0

图 11-14　数据右对齐

对于注入组通道，转换的数据值已经减去了用户在 ADC_JOFRx 寄存器中定义的偏移

量,因此结果可以是一个负值,SEXT 位是扩展的符号值。对于规则组通道,不需减去偏移值,因此只有 12 个位有效。

7. 通道采样时间

ADC 使用若干 ADC_CLK 周期对输入电压采样,采样周期数目可以通过 ADC_SMPRx(x=0、1、2)寄存器中的 SMP[2:0]位更改。每个通道可以分别用不同的时间采样。总的转换时间 TCONV=采样时间+转换时间。每个通道的转换时间由 SMP[2:0]决定,其取值为 4、9、16、24、48、96、192、384 倍的 ADCCLK 周期,如图 11-15 所示。

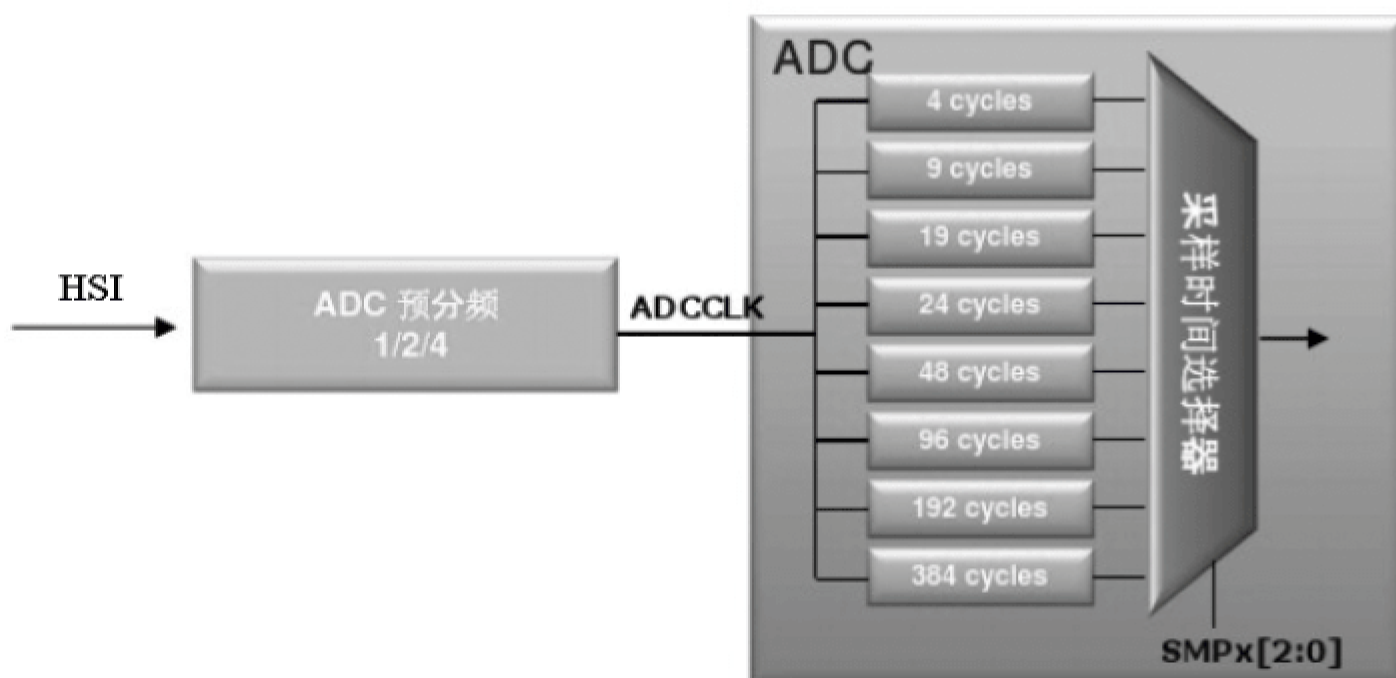


图 11-15　采样周期设置

转换周期与 ADC 的转换位数有关,如图 11-16 所示,当 ADC 分辨率为 12 位时,转换时间为 12 个 ADCCLK 周期,6 位的分辨率时转换时间最短可达 7 个 ADCCLK 周期。因此一个次 ADC 转换的时间例子如下:若 ADCCLK=16MHz,采样时间为 4 个周期,则:

- 12 位分辨率 Tconv = 4+12 = 16 周期=1μs。
- 10 位分辨率 Tconv = 4+11 = 15 周期=937.5ns。
- 8 位分辨率 Tconv = 4+9 = 13 周期=812.5ns。
- 6 位分辨率 Tconv = 4+7 = 11 周期=685ns。

分辨率	转换时间($T_{Conversion}$)
12 bit	12 Cycles
10 bit	11 Cycles
8 bit	9 Cycles
6 bit	7 Cycles

图 11-16　转换时间

这样,我们可以对 ADC 转换通道次序、采样时间和采样次数进行灵活的配置,例如,对 0、2、8、4、7、3、3、3 和 11 通道配置不同的采样时间进行转换,其中通道 3 进行 3 次采样,如图 11-17 所示。

8. 外部触发转换

如图 11-7 下方所示,规则通道、注入通道的转换可以由外部事件触发(比如定时器捕捉、EXTI 线)。如果设置了 EXTEN[1:0]或 JEXTEN[1:0]控制位不为 0,则外部事件就能够触发转换。EXTSEL[3:0]和 JEXTSEL[3:0]控制位允许应用程序选择 16 个可能的事件中的某一个,可以触发规则和注入组的采样。触发信号可以选择上升沿、下降沿或双沿触发。EXTSEL 和 JEXTSEL 的外部事件如表 11-3 所示。

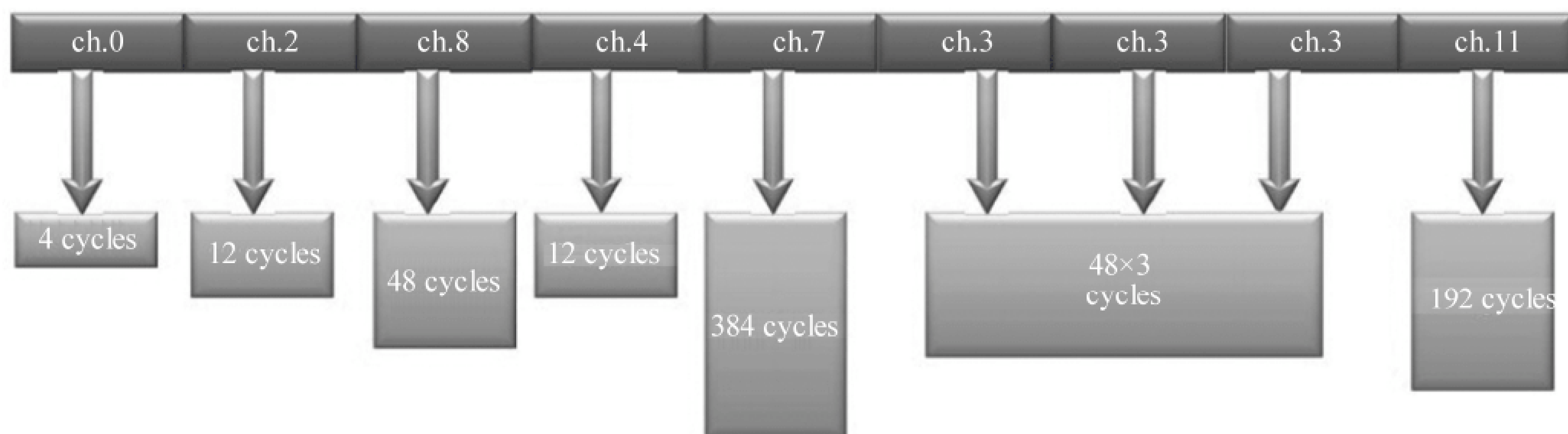


图 11-17　多通道采样时间配置案例

表 11-3　外部触发事件定义

触发事件	类　型	EXTSEL[3：0]	JEXTSEL[3：0]
TIM9_CC1	内部定时器	—	0000
TIM9_CC2	内部定时器	0000	—
TIM9_TRGO	内部定时器	0001	0001
TIM2_CC3	内部定时器	0010	—
TIM2_CC2	内部定时器	0011	—
TIM2_CC1	内部定时器	—	0011
TIM3_TRGO	内部定时器	0100	—
TIM4_CC1	内部定时器	—	0110
TIM4_CC2	内部定时器	—	0111
TIM4_CC3	内部定时器	—	1000
TIM4_CC4	内部定时器	0101	—
TIM2_TRGO	内部定时器	0110	0010
TIM3_CC1	内部定时器	0111	—
TIM3_CC3	内部定时器	1000	—
TIM3_CC4	内部定时器	—	0100
TIM4_TRGO	内部定时器	1001	0101
TIM10_CC1	内部定时器	—	1001
TIM7_TRGO	内部定时器	—	1010
TIM6_TRGO	内部定时器	1010	—
EXTI line11	外部 I/O 引脚	1111	—
EXTI line15	外部 I/O 引脚	—	1111

9. 低速转换的硬件冻结和延迟注入

ADC的转换时钟不采用APB时钟,APB时钟用于MCU和ADC之间的数据访问,当APB时钟太慢,不能满足ADC转换速率时,可以引入延迟以降低转换速率。在每个规则通道和注入组之后插入延迟,在延迟期间,ADC转换的触发信号被忽略。注入组和规则组之间转换时不插入任何延迟,即:

- 如果在规则通道转换期间发生注入,则注入通道转换立即开始;
- 如果被注入通道打断后的规则通道要恢复执行,则立即启动,因为延时已经在上一个规则通道转换完成后添加了。

规则通道的延迟时序如图11-18所示,启用延迟后,在每次规则转换结束之前插入一个延迟,以便CPU在下一个转换完成前有时间读取ADC_DR中的转换数据。延迟的事件长度由ADC_CR2寄存器的DELS[2:0]域指定。

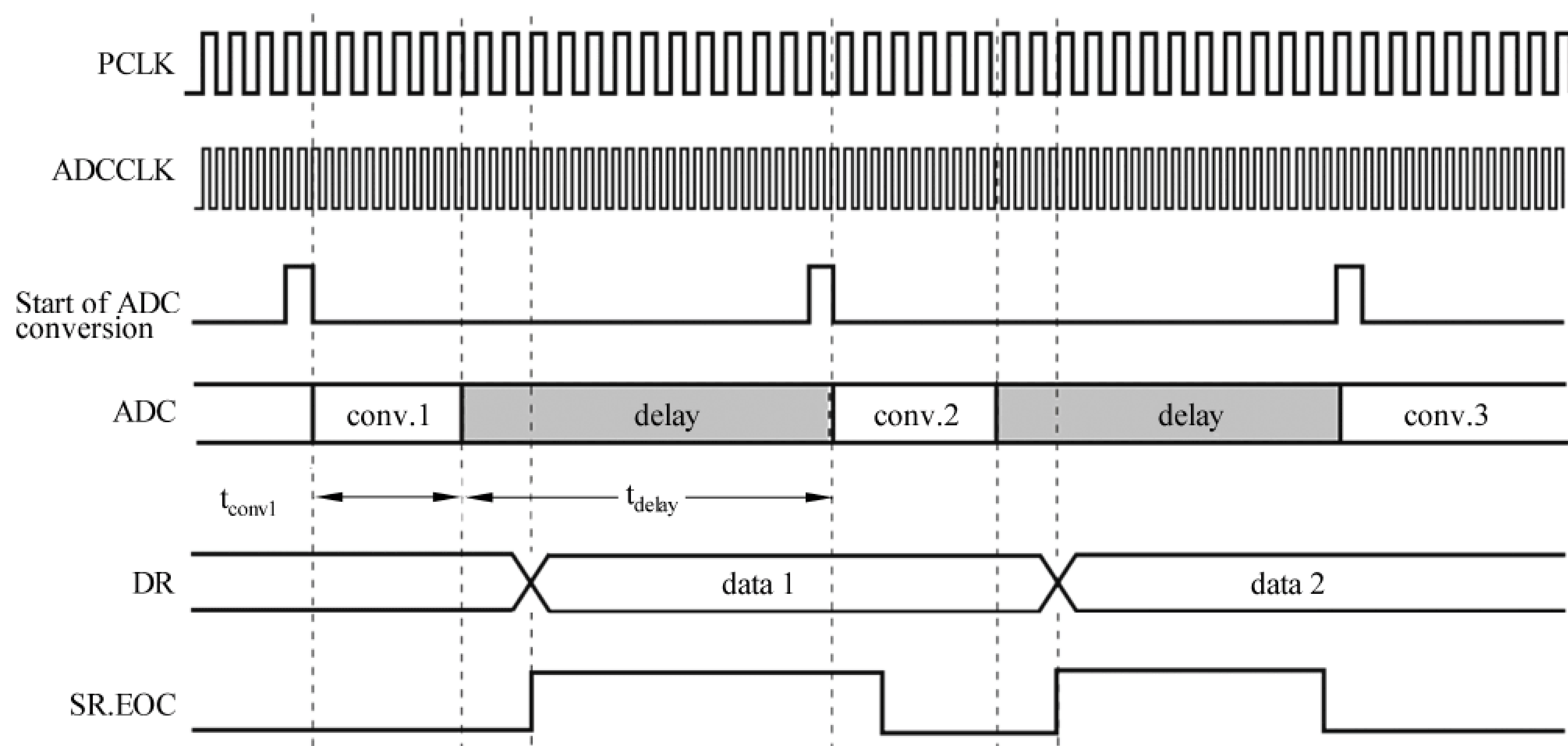


图 11-18 规则通道时延注入

自动注入转换序列后插入延迟的时序如图11-19所示,启用时延后,会在每个注入转换序列的末尾插入延迟。最多可以保存5个ADC转换的数据(1个规则通道ADC_DR和4个注入通道ADC_JDRx),延迟的长度由ADC_CR2寄存器的DELS[2:0]位配置。

配置延迟长度时有两种模式,ADC冻结模式下,即DELS[2:0]=001时,之前通道的所有数据处理完成后才能开始一个新的转换,即规则通道转换,读取ADC_DR寄存器或EOC后位已被清除;注入通道转换,JEOC位被清除时。当配置成ADC延迟插入模式,即DELS[2:0]>001的情况下,新转换只能在上一次转换结束,若干APB周期后才能开始。

10. ADC功耗控制

STM32L152 ADC支持上电和断电管理,以便减少ADC在不进行转换时的功耗,ADC断电的时间包括:

- 注入延迟的时间(当PDD位置1时),当注入延迟结束时,ADC自动通电;
- ADC等待触发事件时(PDI位置1时),ADC在下一次触发事件到达时上电。

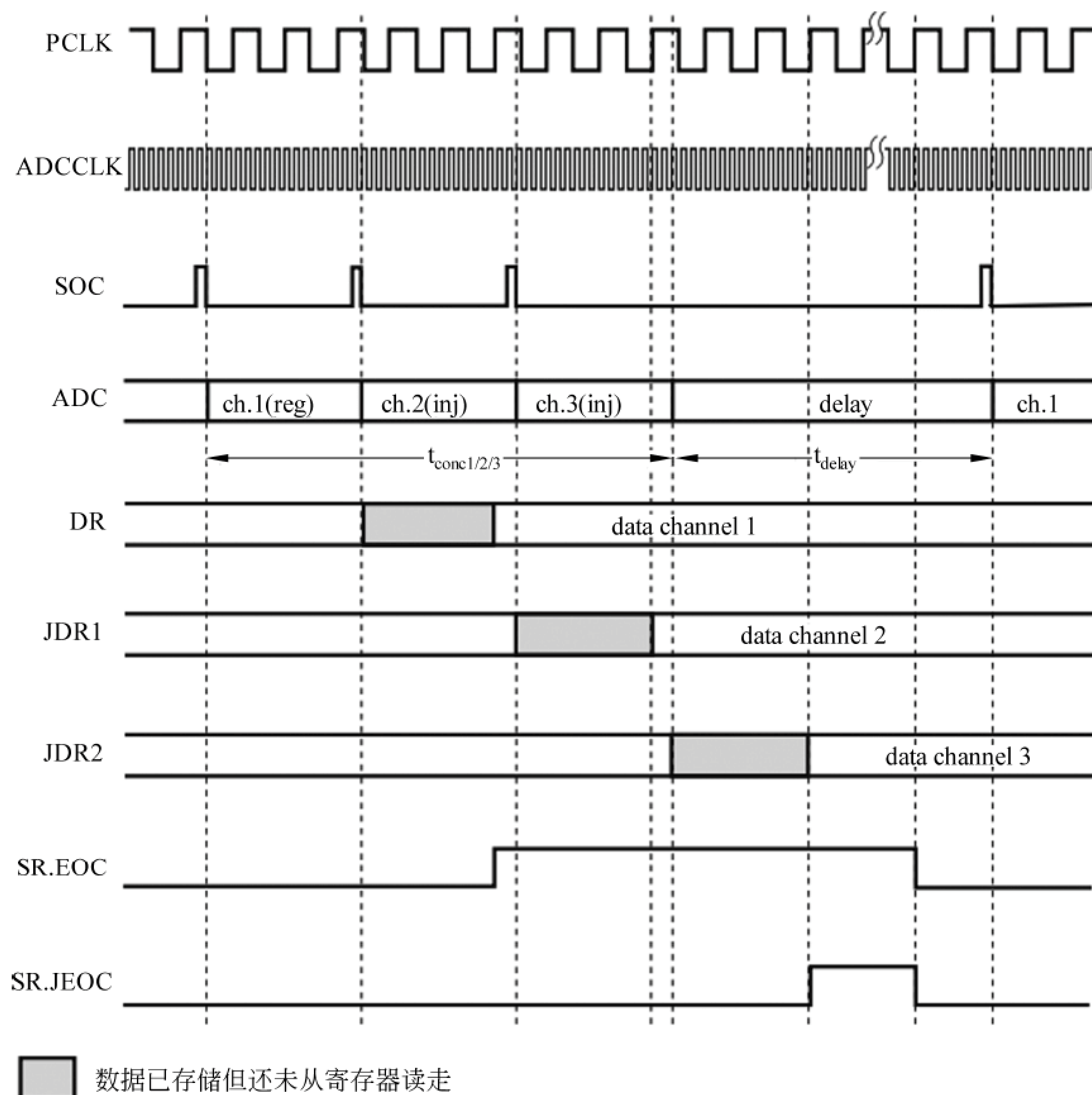


图 11-19　注入通道时延注入

在实际启动转换之前，ADC 需要一定的时间才能启动，在使用自动通断电控制之前，必须考虑到 ADC 上电的启动延迟。因此，采用扫描模式对一组通道进行转换然后断电的方式相比于单个通道转换断电效率更高。对于给定的转换序列，必须在启动之前启用 ADCCLK 时钟直到 EOC 位(或注入时的 JEOC 位)置 1 为止。

图 11-20 为不同情况下的 ADC 断电和上电时序。

11. 模拟看门狗

ADC 的模拟看门狗用于检查电压是否越界，如果被 ADC 转换的模拟电压低于低阈值或高于高阈值，AWD 模拟看门狗状态位被置 1。若 ADC_CR1 寄存器的 AWDIE 位允许产生相应中断则产生模拟看门狗中断。阈值位于 ADC_HTR 和 ADC_LTR 寄存器的最低 12 个有效位中(与对齐模式无关)。

12. 溢出错误检测

溢出检测时钟被启用，只有规则通道转换才会引起溢出错误，在一次 ADC 转换结束时，结果存储在中间缓冲区中直到它被传送到数据寄存器 ADC_DR，如果新的转换数据在

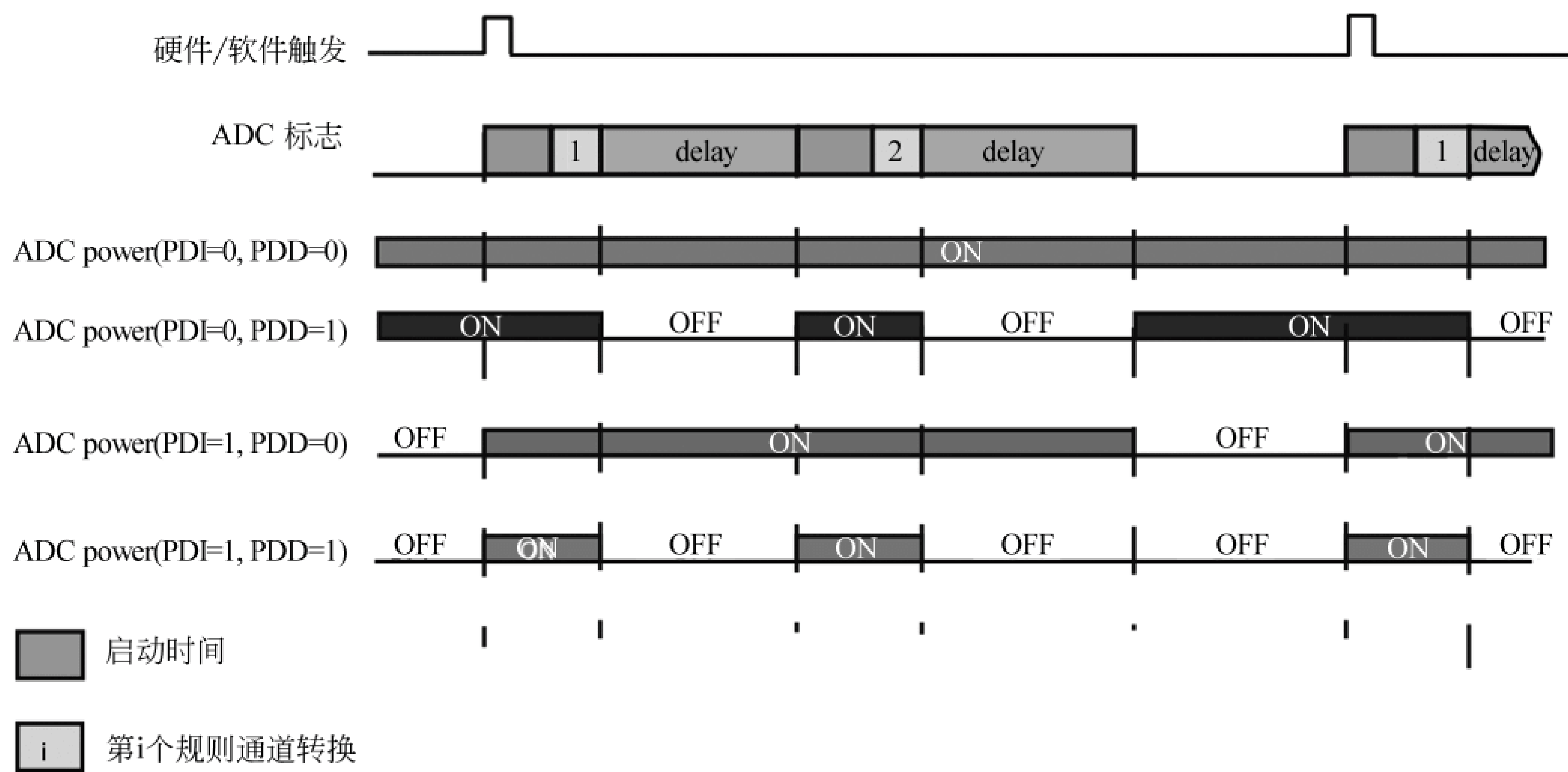


图 11-20 ADC 功耗管理模式

上一个数据传输到 ADC_DR 之前到达，则新数据会被丢弃，检测到溢出错误，ADC_SR 寄存器的 OVR 位置 1，如果 OVRIE 位置 1，则产生溢出中断。以下情况会导致溢出错误：

（1）ADCCLK 时钟和 APB 时钟不匹配，也没有正确注入时延；

（2）ADC_DR 的数据没有及时被读走，导致数据寄存器非空。

13. ADC 中断

ADC 的中断源有多个，如表 11-4 所示。规则组和注入组的转换结束时可以产生中断，当模拟看门狗状态位置 1 或溢出状态位置 1 时，均可产生中断。各种中断源可以独立灵活配置，除此之外，ADC_SR 寄存器还有 5 个状态标志用于管理 ADC，但不能产生中断。

- JCNR（注入通道未准备好）。
- RCNR（规则通道未准备好）。
- ADONS（ADON 状态）。
- JSTRT（注入组通道的转换开始）。
- STRT（规则组通道的转换开始）。

表 11-4 ADC 中断源

中 断 源	中 断 标 记	中断使能位
规则组转换结束	EOC	EOCIE
注入组转换结束	JEOC	JEOCIE
模拟看门狗	AWD	AWDIE
溢出错误	OVR	OVRIE

各种状态和中断的关系如图 11-21 所示。

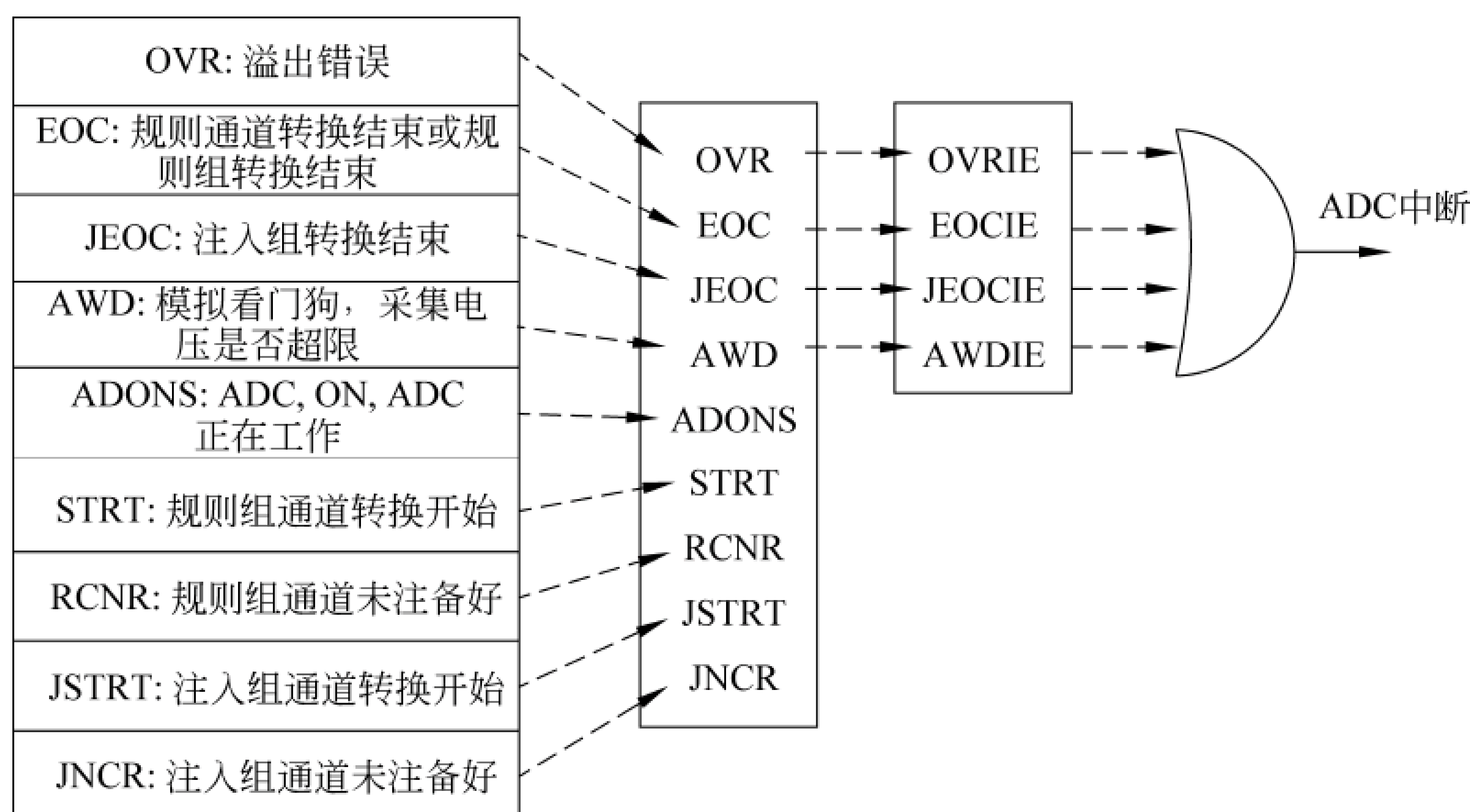


图 11-21　中断和状态标记

11.2.2　温度和电压转换

如图 11-22 所示，ADC 内部两个通道 ADC_IN16 和 ADC_IN17 分别连接到了内置温度传感器和内部参考电源上，可以通过 ADC_CCR 寄存器的 TSVREFFE 位进行使能。

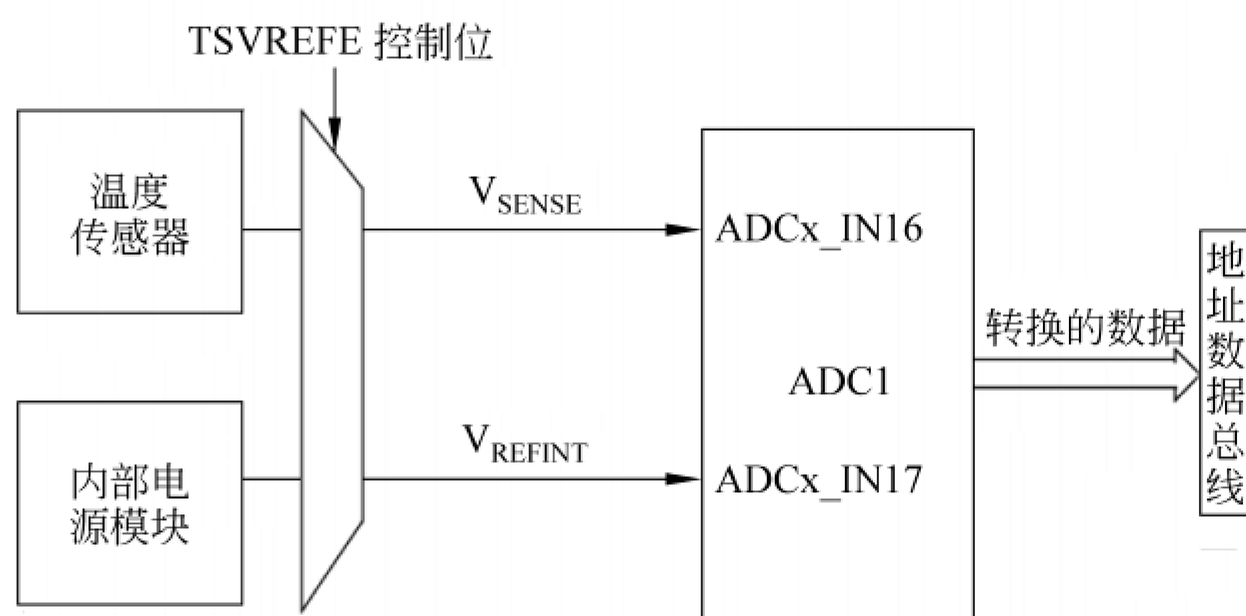


图 11-22　温度和电压测量内部连接通道

温度传感器可以用来测量 MCU 的温度(TA)。温度传感器在内部和 ADC_IN16 输入通道相连接，此通道把传感器输出的电压转换成数字值，不使用时可以将传感器置成掉电模式。温度传感器输出电压随温度线性变化而变化，温度变化曲线的偏移在不同芯片上会有不同，芯片之间温度的测量值可能会有较大差异，ST 在自定义存储区给出了每个芯片的温度的参考测量值 TS_CAL2 和 TS_CAL1，可以通过参考值补偿测量精度。

内部参考电压(VREFINT)为 ADC 和比较器提供一个稳定的参考电压，VREFINT 内部连接到 ADC_IN17 输入通道，该电压位 1.2V，同样，电压的准确值因芯片而异，可以通过读取 ST 自定义存储区的寄存器获得该电压的精确值。

1）读温度的流程

- 选择 ADC1_IN16 输入通道；
- 选择采样时间大于 4μs；
- 设置 ADC_CCR 寄存器的 TSVREFE 位，唤醒掉电模式下的温度传感器；
- 通过设置 ADON 位启动 ADC 转换；
- 读 ADC 数据寄存器 ADC_DR 的数据；
- 利用下列公式得出温度：

$$\text{Temperature}=\frac{110℃-30℃}{\text{TS_CAL2}-\text{TS_CAL1}}\times(\text{TS_DATA}-\text{TS_CAL1})+30℃$$

其中，TS_CAL2 时 110 度时的测量值，TS_CAL1 是 30 度时的测量值，可以通过读取 TS_CAL2 和 TS_CAL1 寄存器获得，TS_DATA 为通道 16 的 ADC 测量值。

2）通过内部参考电压测量系统供电 VDDA

ADC 采样中，微控制器的 VDDA 电源电压可能会发生变化（如电池供电的情况），因此 STM32 内部嵌入了一个参考电压 VREFINT，并将其连接到 ADC_IN17 上，针对每个芯片，其在 VDDA=3V 下参考电压的 ADC 校准值保存在 ST 自定义存储区，由此我们可以通过采样 VREFINT 电压反推 VDDA 电压。实际的 VDDA 电压计算方法为：VDDA = 3 * VREFINT_CAL / VREFINT_DATA，其中，VREFINT_CAL 是 VREFINT 的校准值，VREFINT_DATA 是 ADC_IN17 的采样值。

一般 ADC 采样的范围是 0～VDDA，对于每个 ADC 通道，我们需要将 ADC 采样值转换为实际电压，其转换共识为：

$$V_{\text{CHANNELx}}=\frac{V_{\text{DDA}}}{\text{FULL_SCALE}}\times \text{ADC_DATA}_{\text{x}}$$

其中，FULL_SCALE 指的是 ADC 输出的最大值，跟分辨率有关，例如 12 比特的分辨率，FULL_SCALE = $2^{12}-1=4095$。

结合上述 VDDA 的计算和校准方法，我们可以得到每个通道的实际电压的校准公式：

$$V_{\text{CHANNELx}}=\frac{3\text{V}\times\text{VREFINT_CAL}\times\text{ADC_DATA}_{\text{x}}}{\text{VREFINT_DATA}\times\text{FULL_SCALE}}$$

11.3 ADC 寄存器

ADC 涉及的寄存器如表 11-5 所示。

表 11-5　ADC 寄存器

寄存器名称	地址偏移量	作　　用	默认值
ADC_SR	0x00	ADC 状态标志	0x00000000
ADC_CR1	0x04	控制寄存器，设置扫描模式、中断允许等	0x00000000

续表

寄存器名称	地址偏移量	作　　用	默认值
ADC_CR2	0x08	控制寄存器，设置数据对齐方式、转换模式等	0x00000000
ADC_SMPR1～ADC_SMPR3	0x0c～0x14	配置 ADC 各通道的采样时间	0x00000000
ADC_JOFR1～ADC_JOFR4	0x18～0x24	配置 ADC 注入通道数据偏移量	0x00000000
ADC_HTR	0x28	模拟看门狗超限高阈值	0x00000000
ADC_LTR	0x2c	模拟看门狗超限低阈值	0x00000000
ADC_SQR1～ADC_SQR5	0x30～0x40	规则通道的数量和通道序列	0x00000000
ADC_JSQR	0x44	注入通道的数量和通道序列	0x00000000
ADC_JDR1～ADC_JDR4	0x48～0x54	存储注入通道转换数据	0x00000000
ADC_DR	0x58	存储规则通道转换数据	0x00000000
ADC_SMPR0	0x5c	ADC_IN30 和 ADC_IN31 采样时间	0x00000000
ADC_CSR	0x300	ADC_SR 的镜像	0x00000000
ADC_CCR	0x304	配置分频及内部温度、电压使能	0x00000000

1. ADC 状态寄存器 ADC_SR

ADC 状态寄存器用于标记 ADC 转换过程中的状态量，其有效域定义如图 11-23 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Reserved						JCNR	RCNR	Res.	ADONS	OVR	STRT	JSTRT	JEOC	EOC	AWD
						r	r		r	rc_w0	rc_w0	rc_w0	rc_w0	rc_w0	rc_w0

图 11-23　ADC 状态寄存器

JCRN：注入通道未就绪，该位在 JSQR 寄存器被写后由硬件设置或清除，0 表示注入通道未就绪，1 表示就绪。

RCNR：规则通道未就绪，该位在 SQRx 寄存器被写后由硬件设置或清除，0 表示注入规则通道未就绪，1 表示就绪。

ADONS：ADC 开启状态，该位由硬件设置或清除，用于表示 ADC 是否准备好可以开始转换；0 表示 ADC 还未准备好，1 表示 ADC 可以开始一个转换。

OVR：溢出错误标记，该位为 1 表示有溢出错误产生，0 表示没有溢出错误；当规则通道转换数据丢失时，该位由硬件自动置 1，可以通过软件清 0。

STRT：规则通道开始位，该位由硬件在规则通道转换开始时设置，由软件清除。0 表示规则通道转换未开始；1 表示规则通道转换已开始。

JSTRT：注入通道开始位，该位由硬件在注入通道组转换开始时设置，由软件清除。0

表示注入通道组转换未开始，1 表示注入通道组转换已开始。

JEOC：注入通道转换结束位，该位由硬件在所有注入通道组转换结束时设置，由软件清除，0 表示转换未完成，1 表示转换完成。

EOC：转换结束位，该位由硬件在(规则或注入)通道组转换结束时设置，由软件清除或由读取 ADC_DR 时清除，0 表示转换未完成，1 表示转换完成。

AWD：模拟看门狗标志位，该位由硬件在转换的电压值超出了 ADC_LTR 和 ADC_HTR 寄存器定义的范围时设置，由软件清除，0 表示没有发生模拟看门狗事件，1 表示发生模拟看门狗事件。

2. ADC 控制寄存器 ADC_CR1

控制寄存器 ADC_CR1 用于设置扫描模式、中断允许等配置，其有效域定义如图 11-24 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved					OVRIE	RES[1:0]		AWDEN	JAWDEN	Reserved				PDI	PDD
					rw	rw	rw	rw	rw					rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DISCNUM[2:0]			JDISCEN	DISCEN	JAUTO	AWDSGL	SCAN	JEOCIE	AWDIE	EOCIE	AWDCH[4:0]				
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 11-24　控制寄存器 ADC_CR1

OVRIE：溢出错误中断使能，1 表示启用溢出中断，当 OVR＝1 时，产生一个 ADC 中断，0 表示禁用溢出中断。

RES[1：0]：分辨率设置，00～11 分别表示 12 位、10 位、8 位和 6 位的分辨率，这些位必须在 ADON＝1 时配置。

AWDEN：在规则通道上开启模拟看门狗，0 表示禁用模拟看门狗，1 表示使用。

JAWDEN：在注入通道上开启模拟看门狗，0 表示禁用模拟看门狗，1 表示使用。

PDI：空闲期间掉电，当 ADON＝1，该位置 1 表示在没有转换的时候 ADC 掉电，等待下一个触发事件，否则为不断电。

PDD：延迟期间掉电，当 ADON＝1 时，该位置 1 表示在两个转换之间注入的延迟期间，ADC 断电，否则为不断电。

DISCNUM[2：0]：间断模式通道计数，软件通过这些位定义在间断模式下，收到外部触发后转换规则通道的数目，编码 000～111 分别表示 1～8 个通道。

JDISCEN：开启或关闭注入通道组的间断模式，0 表示禁用间断模式，1 表示允许。

DISCEN：开启或关闭规则通道组上的间断模式，0 表示禁用间断模式，1 表示允许。

JAUTO：开启或关闭规则通道组转换结束后自动注入通道组转换，0 表示关闭，1 表示开启。

AWDSGL：开启或关闭由 AWDCH[4：0]位指定的通道上的模拟看门狗功能，0 表示在所有的通道上使用模拟看门狗，1 表示在单一通道上使用模拟看门狗。

SCAN：开启或关闭扫描模式，在扫描模式中，转换通道来自 ADC_SQRx 或 ADC_JSQRx 寄存器，0 表示关闭扫描模式，1 表示使用扫描模式。该位只能在 ADON＝0 时设置。如果分别设置了 EOCIE 或 JEOCIE 位，只在最后一个通道转换完毕后才会产生 EOC 或 JEOC 中断。JEOCIE：用于禁止或允许所有注入通道转换结束后产生中断。1 表示允许 JEOC 中断，当硬件设置 JEOC 位时产生中断。

AWDIE：用于禁止或允许模拟看门狗产生中断，0 表示禁止，1 表示允许。在扫描模式下，如果看门狗检测到超范围的数值时，如果设置该位，中止 AD 转换。

EOCIE：用于禁止或允许转换结束后产生中断，0 表示禁止，1 表示允许，当硬件设置 EOC 位时产生中断。

AWDCH[4：0]：用于选择模拟看门狗保护的输入通道，00000～11010 分别表示 ADC 模拟输入通道 0～ADC 模拟输入通道 26。

3. ADC 控制寄存器 ADC_CR2

控制寄存器 ADC_CR2 用于设置数据对齐方式、连续转换位、ADC 启动位、外部触发转换等，其有效域定义如图 11-25 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	SWSTART	EXTEN		EXTSEL[3:0]				Res.	JSWSTART	JEXTEN		JEXTSEL[3:0]			
	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved				ALIGN	EOCS	DDS	DMA	Res.	DELS			Res.	ADC_CFG	CONT	ADON
				rw	rw	rw	rw		rw	rw	rw		rw	rw	rw

图 11-25　控制寄存器 ADC_CR2

SWSTART：开始转换规则通道，由软件设置该位以启动转换，转换开始后硬件马上清除此位。如果在 EXTSEL[2：0]位中选择了 SWSTART 为触发事件，该位用于启动一组规则通道的转换，0 表示复位状态，1 表示开始转换规则通道。

EXTEN[1：0]：规则通道外部触发使能，由软件设置或清除，用于配置外部触发的方式，00 表示禁用外部触发，01 表示上升沿触发，10 表示下降沿触发，11 表示双沿触发，该域只能在 ADONS＝1 时配置为有效。

EXTSEL[3：0]：规则通道外部触发源选择，具体配置见表 11-3。

JSWSTART：开始转换注入通道，由软件设置该位以启动转换，软件可清除此位或在转换开始后硬件马上清除此位。如果在 JEXTSEL[2：0]位中选择了 JSWSTART 为触发事件，该位用于启动一组注入通道的转换，0 表示复位状态，1 表示开始转换注入通道。

JEXTEN[1：0]：注入通道外部触发使能，由软件设置或清除，用于配置外部触发的方式，00 表示禁用外部触发，01 表示上升沿触发，10 表示下降沿触发，11 表示双沿触发，该域只能在 ADONS＝1 时配置为有效。

JEXTSEL[3：0]：注入通道外部触发源选择，具体配置见表 11-3。

ALIGN：数据对齐，0 表示右对齐，1 表示左对齐。

EOCS：EOC 标记置位选择，0 表示在所有规则组通道转换完成后 EOC 置 1，1 表示规则组的每个通道转换完后 EOC 置 1。

DDS：DMA 请求选择，0 表示上一个数据转换完后不产生 DMA 请求，1 表示只要数据转换，就发起 DMA 请求。

DMA：直接存储器访问模式使能，0 表示不使用 DMA 模式，1 表示使用 DMA 模式。

DELS[2：0]：时延选择。该域用于选择所要注入的延迟的长度。000 表示不注入延迟，001 表示延迟的时间为直到转换完的数据被读走（规则通道的读数据寄存器或 EOC＝0，注入通道的 JEOC＝0），010-111 分别表示延迟时长为：7、15、21、63、127、255 个 APB 的时钟周期。在使用 DELS 配置时注意，延迟的最小值为：如果 APB 的时钟小于 ADC 时钟的 1/2，最少 15 个 APB 周期；如果 APB 时钟大于 ADC 时钟的 1/2 但小于 ADC 时钟，则最少为 7 个 APB 周期。

ADC_CFG：用于选择 ADC 的配置，使用 A 面或 B 面，0 表示使用 A 面，对应的 ADC 通道为 ADC_IN0 ～ ADC_IN31；1 表示使用通道 B 面，对应的 ADC 输入通道为 ADC_IN0b ～ ADC_IN31b。

CONT：连续转换，如果设置了此位，则转换将连续进行直到该位被清除，0 表示单次转换模式，置 1 表示连续转换模式。

ADON：A/D 转换器开关，当该位为 0 时，写入 1 将把 ADC 从断电模式下唤醒；当该位为 1 时，写入 1 将启动转换。需注意，在转换器上电至转换开始有一个延迟 tSTAB，具体值参考 STM32L152 数据手册。

4. ADC 采样时间寄存器 ADC_SMPRx（x＝1，2，3，4）

ADC_SMPRx 寄存器用于设置 ADC 各通道的采样时间，共有四个寄存器，ADC_SMPR4 寄存器的有效域定义如图 11-26 所示。每个通道使用 3 位，每个寄存器可以表示 9 个通道。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved		SMP9[2:0]			SMP8[2:0]			SMP7[2:0]			SMP6[2:0]			SMP5[2:1]	
		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SMP5[0]	SMP4[2:0]			SMP3[2:0]			SMP2[2:0]			SMP1[2:0]			SMP0[2:0]		
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 11-26　ADC_SMPR4 寄存器

SMPx[2：0]：用于独立地选择每个通道的采样时间。在采样周期中通道选择位必须保持不变。000-111 分别表示 4、9、16、24、48、96、192、384 个 ADC 时钟周期。

5. ADC 注入通道数据偏移寄存器 ADC_JOFRx（x＝1，2，3，4）

注入通道数据偏移寄存器用于设置 ADC 注入通道数据偏移量，共有 4 个注入通道，因此有 4 个寄存器，寄存器的有效域定义如图 11-27 所示。

JOFFSETx[11：0]：注入通道 x 的数据偏移，当转换注入通道时，这些位定义了用于从原始转换数据中减去的数值。转换的结果可以在 ADC_JDRx 寄存器中读出。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved				JOFFSETx[11:0]											
				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 11-27　注入通道数据偏移寄存器

6. ADC 看门狗高/低阈值寄存器(ADC_HTR、ADC_LRT)

看门狗阈值寄存器包括高阈值寄存器 ADC_HTR 和低阈值寄存器 ADC_LTR,用于设置 ADC 模拟看门狗的高低阈值。两个寄存器的有效域定义如图 11-28 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved				HT[11:0]											
				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved				LT[11:0]											
				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 11-28　看门狗阈值寄存器

HT[11:0]和 LT[11:0]分别用于定义模拟看门狗的阈值高限和低限。

7. ADC 规则通道序列寄存器 ADC_SQRx(x=1,2,3,4,5)

规则通道序列寄存器用于设置规则通道序列长度、对应序列中各个转换的通道编号(最多 28 个)。该寄存器有 5 个,每个通道占用 5 位用于表示一个通道编号,其中 ADC_SQR1 寄存器还有一个 L[4:0]域,其余的 4 个寄存器只有 SQx[4:0]域。ADC_SQR1 的有效域定义如图 11-29 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved							L[4:0]					SQ28[4:1]			
							rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SQ28[0]	SQ27[4:0]					SQ26[4:0]					SQ25[4:0]				
rw	rw	rw	rw	rw	rw	rw	rw				rw	rw	rw	rw	rw

图 11-29　规则通道序列寄存器 1

L[4:0]:规则通道序列的长度,用于指定总共配置了多少个通道,00000～11011 分别表示 1、2、… 28 个通道。

SQx[4:0]:规则序列中的第 x 个转换的通道编号。

8. ADC 注入序列寄存器 ADC_JSQR

注入序列寄存器用于设置注入通道序列长度、对应序列中各个转换的通道编号(最多 4 个),其有效域定义如图 11-30 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved										JL[1:0]		JSQ4[4:1]			
										rw	rw	rw	rw	rw	rw
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
JSQ4[0]	JSQ3[4:0]					JSQ2[4:0]					JSQ1[4:0]				
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 11-30　注入序列寄存器

JL[1：0]：注入通道序列长度，用于指定注入通道转换序列中的通道数目，00～11 分别表示 1～4 个通道。

JSQx[4：0]：注入序列中的第 x 个转换通道的编号。

如果 JL[1：0]的长度小于 4，则转换的序列顺序是从(4－JL)开始。例如：JL＝3，则转换序列是 JSQ1[4：0]、JSQ2[4：0]、JSQ3[4：0]、JSQ4[4：0]；如果 JL＝2，则 ADC 的转换序列是 JSQ2[4：0]、JSQ3[4：0]、JSQ4[4：0]。

9. ADC 注入数据寄存器 ADC_JDRx（x＝1，2，3，4）

注入数据寄存器用于存放 ADC 注入通道转换后的数据，最多有 4 个注入通道，因此有 4 个注入数据寄存器，其有效域定义如图 11-31 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
JDATA[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

图 11-31　注入数据寄存器

JDATA[15：0]：注入转换的数据，只读，存储注入通道的转换结果。数据是左对齐或右对齐。

10. ADC 规则数据寄存器 ADC_DR

规则数据寄存器用于存放 ADC 规则通道转换后的数据，所有规则通道共享一个数据寄存器，其有效域定义如图 11-32 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
DATA[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

图 11-32　规则数据寄存器

DATA[15：0]：规则转换的数据，只读，包含了规则通道的转换结果。数据是左对齐或右对齐。

11. ADC 采样时间寄存器 ADC_SMPR0

采样时间寄存器 0 用于配置通道 ADC 通道 30 和通道 31 的采样时间，不是所有的 STM32L152 系列都有 30 和 31 通道。

12. ADC 通用状态寄存器 ADC_CSR

通用状态寄存器提供 ADC 状态寄存器的镜像，其有效域定义如图 11-33 所示，所有域只读且不允许清除，而是通过对 ADC_SR 寄存器的相应位写 0 来清除。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved									ADONS1	OVR1	STRT1	JSTRT1	JEOC 1	EOC1	AWD1
									r	r	r	r	r	r	r

图 11-33　通用状态寄存器

ADONS1：ADC_SR 寄存器中 ADONS 位的副本。

OVR1：ADC_SR 寄存器中 OVR 位的副本。

STRT1：ADC_SR 寄存器中 STRT 位的副本。

JSTRT1：ADC_SR 寄存器中 JSTRT 位的副本。

JEOC1：ADC_SR 寄存器中 JEOC 位的副本。

EOC1：ADC_SR 寄存器中 EOC 位的副本。

AWD1：ADC_SR 寄存器中 AWD 位的副本。

13. ADC 通用控制寄存器 ADC_CCR

该寄存器用于配置 ADC 的采样时钟预分频和内部温度、电压使能，其有效域定义如图 11-34 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								TSVREFE	Reserved					ADCPRE[1:0]	
								rw						rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved															

图 11-34　通用控制寄存器

TSVREFE：用于开启或禁止温度传感器和 VREFINT 通道。0 表示禁止，1 表示启用。

ADCPRE[1：0]：ADC 的预分频系数，00 表示 HSI 不分频，01 表示 2 分频，10 表示 4 分频，11 保留。

11.4　ADC 寄存器结构及 ADC 库函数

ADC 寄存器结构描述了固件函数库所使用的数据结构，固件库函数介绍了 ST 提供的典型库函数。

11.4.1 ADC寄存器结构

ADC寄存器结构,ADC_TypeDeff和ADC_Common_TypeDef在文件stm32l1xx.h中定义如下:

```
typedef struct
{
  __IO uint32_t SR;
  __IO uint32_t CR1;
  __IO uint32_t CR2;
  __IO uint32_t SMPR1;
  __IO uint32_t SMPR2;
  __IO uint32_t SMPR3;
  __IO uint32_t JOFR1;
  __IO uint32_t JOFR2;
  __IO uint32_t JOFR3;
  __IO uint32_t JOFR4;
  __IO uint32_t HTR;
  __IO uint32_t LTR;
  __IO uint32_t SQR1;
  __IO uint32_t SQR2;
  __IO uint32_t SQR3;
  __IO uint32_t SQR4;
  __IO uint32_t SQR5;
  __IO uint32_t JSQR;
  __IO uint32_t JDR1;
  __IO uint32_t JDR2;
  __IO uint32_t JDR3;
  __IO uint32_t JDR4;
  __IO uint32_t DR;
  __IO uint32_t SMPR0;
} ADC_TypeDef;
typedef struct
{
  __IO uint32_t CSR;
  __IO uint32_t CCR;
} ADC_Common_TypeDef;
```

ADC外设声明于文件stm32l1xx.h:

```
#define PERIPH_BASE                ((uint32_t)0x40000000)
#define APB1PERIPH_BASE PERIPH_BASE
```

```
#define APB2PERIPH_BASE              (PERIPH_BASE + 0x10000)
#define AHBPERIPH_BASE               (PERIPH_BASE + 0x20000)
#define ADC1_BASE                    (APB2PERIPH_BASE + 0x2400)
#define ADC_BASE                     (APB2PERIPH_BASE + 0x2700)
#define ADC1                         ((ADC_TypeDef * ) ADC1_BASE)
#define ADC                          ((ADC_Common_TypeDef * ) ADC_BASE)
```

ADC_InitTypeDef 和 ADC_CommonInitTypeDef 定义于文件 stm32l1xx_adc.h，用于寄存器的初始化。

```
typedef struct
{
  uint32_t ADC_Resolution;                          //分辨率
  FunctionalState ADC_ScanConvMode;                 //是否扫描模式
  FunctionalState ADC_ContinuousConvMode;           //是否连续转换
  uint32_t ADC_ExternalTrigConvEdge;                //外部触发信号边沿
  uint32_t ADC_ExternalTrigConv;                    //外部触发信号
  uint32_t ADC_DataAlign;                           //对其模式
  uint8_t  ADC_NbrOfConversion;                     //转换通道数量
}ADC_InitTypeDef;
typedef struct
{
  uint32_t ADC_Prescaler;                           //分频系数
}ADC_CommonInitTypeDef;
```

ADC_Resolution 用于指定转换的分辨率，其取值为：ADC_Resolution_12b、ADC_Resolution_10b、ADC_Resolution_8b、ADC_Resolution_6b，分别表示 12 位、10 位、8 位和 6 位分辨率。

ADC_ScanConvMode 用于指定是否启用扫描模式，其取值为：ENABLE 和 DISABLE。

ADC_ContinuousConvMode 用于指定是否是连续转换，其取值为：ENABLE 和 DISABLE。

ADC_ExternalTrigConvEdge 用于指定外部触发信号的边沿，其取值为：ADC_ExternalTrigConvEdge_None、ADC_ExternalTrigConvEdge_Rising、ADC_ExternalTrigConvEdge_Falling、ADC_ExternalTrigConvEdge_RisingFalling

ADC_ExternalTrigConv 用于指定外部触发信号，规则组的触发信号定义为：

- ADC_ExternalTrigConv_T2_CC3
- ADC_ExternalTrigConv_T2_CC2
- ADC_ExternalTrigConv_T2_TRGO
- ADC_ExternalTrigConv_T3_CC1
- ADC_ExternalTrigConv_T3_CC3

- ADC_ExternalTrigConv_T3_TRGO
- ADC_ExternalTrigConv_T4_CC4
- ADC_ExternalTrigConv_T4_TRGO
- ADC_ExternalTrigConv_T6_TRGO
- ADC_ExternalTrigConv_T9_CC2
- ADC_ExternalTrigConv_T9_TRGO
- ADC_ExternalTrigConv_Ext_IT11

注入组的触发信号为：

- ADC_ExternalTrigInjecConv_T2_TRGO
- ADC_ExternalTrigInjecConv_T2_CC1
- ADC_ExternalTrigInjecConv_T3_CC4
- ADC_ExternalTrigInjecConv_T4_TRGO
- ADC_ExternalTrigInjecConv_T4_CC1
- ADC_ExternalTrigInjecConv_T4_CC2
- ADC_ExternalTrigInjecConv_T4_CC3
- ADC_ExternalTrigInjecConv_T7_TRGO
- ADC_ExternalTrigInjecConv_T9_CC1
- ADC_ExternalTrigInjecConv_T9_TRGO
- ADC_ExternalTrigInjecConv_T10_CC1
- ADC_ExternalTrigInjecConv_Ext_IT15

ADC_DataAlign 用于指定数据对齐模式，其取值为：ADC_DataAlign_Right、ADC_DataAlign_Left。

ADC_Prescaler 用于指定 ADC 采样时钟的预分频系数，其取值为：ADC_Prescaler_Div1、ADC_Prescaler_Div2、ADC_Prescaler_Div4。

11.4.2 ADC 库函数

ST 提供的 ADC 库函数如表 11-6 所示。

表 11-6 ADC 库函数

函　数	功　能
ADC_DeInit	初始化寄存器为复位值
ADC_Init	根据初始化模板 ADC_InitTypeDef 初始化 ADC 寄存器
ADC_StructInit	将 ADC_InitTypeDef 初始化变量设为默认值
ADC_CommonInit	根据初始化模板 ADC_CommonInitTypeDef 初始化 ADC 寄存器

续表

函 数	功 能
ADC_CommonStructInit	将 ADC_CommonInitTypeDef 初始化变量设为默认值
ADC_Cmd	启动或停止 ADC
ADC_BankSelection	ADC 输入引脚 A、B 面选择
ADC_PowerDownCmd	断电控制
ADC_DelaySelectionConfig	时延配置
ADC_AnalogWatchdogCmd	启用或停止模拟看门狗
ADC_AnalogWatchdogThresholdsConfig	模拟看门狗高低阈值配置
ADC_AnalogWatchdogSingleChannelConfig	模拟看门狗通道配置
ADC_TempSensorVrefintCmd	使能内部温度和参考电压通道
ADC_RegularChannelConfig	规则组通道配置
ADC_SoftwareStartConv	启动转换
ADC_GetSoftwareStartConvStatus	获取转换状态
ADC_EOCOnEachRegularChannelCmd	配置每次规则组通道产生 EOC
ADC_ContinuousModeCmd	连续转换模式
ADC_DiscModeChannelCountConfig	间断模式通道数量配置
ADC_DiscModeCmd	使能间断模式
ADC_GetConversionValue	获取转换结果
ADC_DMACmd	启用 DMA
ADC_DMARequestAfterLastTransferCmd	配置 DMA 的产生时机
ADC_InjectedChannelConfig	注入通道配置
ADC_InjectedSequencerLengthConfig	注入通道序列长度设置
DC_SetInjectedOffset	注入通道偏移量
ADC_ExternalTrigInjectedConvConfig	注入通道外部触发配置
ADC_ExternalTrigInjectedConvEdgeConfig	诸如通道外部触发信号边沿设置
ADC_SoftwareStartInjectedConv	启动注入通道转换
ADC_GetSoftwareStartInjectedConvCmd	获取注入通道转换状态
ADC_AutoInjectedConvCmd	启用自动注入
ADC_InjectedDiscModeCmd	使能注入通道间断模式
ADC_GetInjectedConversionValue	获取注入通道转换值
ADC_ITConfig	配置 ADC 中断源

续表

函　　数	功　　能
ADC_GetFlagStatus	获取状态标志结果
ADC_ClearFlag	清除状态位
ADC_GetITStatus	获取中断源
ADC_ClearITPendingBit	清除中断源

1）函数 ADC_DeInit

函数功能：初始化 ADC1 寄存器为复位值。

函数原型：void ADC_DeInit(ADC_TypeDef * ADCx)。

输入参数 ADCx：需要初始化的 ADC 外设，取值为 ADC1。

2）函数 ADC_Init

函数功能：根据 ADC_InitStruct 指定的参数初始化 ADC1 寄存器。

函数原型：void ADC_Init(ADC_TypeDef * ADCx，ADC_InitTypeDef * ADC_InitStruct)。

输入参数 ADCx：需要初始化的 ADC 外设，取值为 ADC1。

输入参数 ADC_InitStruct：指向 ADC_InitTypeDef 结构体的指针。

3）函数 ADC_StructInit

函数功能：使用默认值初始化 ADC_InitStruct 成员。

函数原型：void ADC_StructInit(ADC_InitTypeDef * ADC_InitStruct)。

输入参数 ADC_InitStruct：指向 ADC_InitTypeDef 结构的指针。

默认值为：

```
ADC_Resolution = ADC_Resolution_12b;
ADC_ScanConvMode = DISABLE;
ADC_ContinuousConvMode = DISABLE;
ADC_ExternalTrigConvEdge = ADC_ExternalTrigConvEdge_None;
ADC_ExternalTrigConv = ADC_ExternalTrigConv_T2_CC2;
ADC_DataAlign = ADC_DataAlign_Right;
ADC_NbrOfConversion = 1;
```

4）函数 ADC_CommonInit

函数功能：根据 ADC_CommonInitStruct 指定的参数初始化 ADC 外设。

函数原型：void ADC_CommonInit(ADC_CommonInitTypeDef * ADC_CommonInitStruct)。

输入参数 ADC_CommonInitStruct：指向 ADC_CommonInitTypeDef 结构的指针。

5）函数 ADC_CommonStructInit

函数功能：使用默认值初始化 ADC_CommonInitStruct 成员。

函数原型：void ADC_CommonStructInit(ADC_CommonInitTypeDef * ADC_CommonInitStruct)。

输入参数 ADC_CommonInitStruct：指向 ADC_CommonInitTypeDef 结构的指针。

默认值 ADC_Prescaler = ADC_Prescaler_Div1。

6）函数 ADC_Cmd

函数功能：启用或停止 ADC。

函数原型：void ADC_Cmd(ADC_TypeDef * ADCx，FunctionalState NewState)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

输入参数 NewState：ADC 的状态，取值为 ENABLE 或 DISABLE。

7）函数 ADC_BankSelection

函数功能：选择 ADC 输入通道来源。

函数原型：void ADC_BankSelection(ADC_TypeDef * ADCx，uint8_t ADC_Bank)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

输入参数 ADC_Bank：要选择的 ADC 输入通道，取值为 ADC_Bank_A 和 ADC_Bank_B，当选用 ADC_Bank_A 时，ADC 通道为 ADC_IN0～ADC_IN31，当选用 ADC_Bank_B 时，ADC 通道为 ADC_IN0b～ADC_IN31b。

8）函数 ADC_PowerDownCmd

函数功能：在注入延迟或空闲期间启用或禁用 ADC 掉电功能。

函数原型：void ADC_PowerDownCmd(ADC_TypeDef * ADCx，uint32_t ADC_PowerDown，FunctionalState NewState)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

输入参数 ADC_PowerDown：ADC 掉电配置，取值范围为：

- ADC_PowerDown_Delay：ADC 在延迟阶段断电。
- ADC_PowerDown_Idle：ADC 在空闲阶段关闭。
- ADC_PowerDown_Idle_Delay：ADC 在延迟和空闲阶段断电。

输入参数 NewState：ADCx 掉电的新状态，取值为 ENABLE 或 DISABLE。

9）函数 ADC_DelaySelectionConfig

函数功能：配置 ADC 注入延迟的大小。

函数原型：void ADC_DelaySelectionConfig(ADC_TypeDef * ADCx，uint8_t ADC_DelayLength)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

输入参数 ADC_DelayLength：注入的延迟长度，取值为：

- ADC_DelayLength_None：没有延迟；
- ADC_DelayLength_Freeze：延迟到读取转换后的数据；
- ADC_DelayLength_7Cycles：延迟长度等于 7 个 APB 时钟周期；
- ADC_DelayLength_15Cycles：延迟长度等于 15 个 APB 时钟周期；
- ADC_DelayLength_31Cycles：延迟长度等于 31 个 APB 时钟周期；
- ADC_DelayLength_63Cycles：延迟长度等于 63 个 APB 时钟周期；
- ADC_DelayLength_127Cycles：延迟长度等于 127 个 APB 时钟周期；

- ADC_DelayLength_255Cycles：延迟长度等于 255 个 APB 时钟周期。

10）函数 ADC_TempSensorVrefintCmd

函数功能：启用/禁用 ADC 与温度传感器和 Vrefint 源之间的内部连接。

函数原型：void ADC_TempSensorVrefintCmd(FunctionalState NewState)。

输入参数 NewState：温度传感器和 Vrefint 是否连接，取值为 ENABLE 或 DISABLE。

11）函数 ADC_RegularChannelConfig

函数功能：为所选 ADC 规则通道配置其对应的规则组序号及其采样时间。

函数原型：void ADC_RegularChannelConfig(ADC_TypeDef * ADCx，uint8_t ADC_Channel，uint8_t Rank，uint8_t ADC_SampleTime)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

输入参数 ADC_Channel：要配置的 ADC 通道，取值为 ADC_Channel_x(x=0～27)或 ADC_Channel_xb(x=0～12)。

输入参数 Rank：所要配置的通道在规则组通道的次序，取值范围为 1～28。

输入参数 ADC_SampleTime：所选择通道的采样时间，取值范围为：

- ADC_SampleTime_4Cycles：采样时间等于 4 个周期；
- ADC_SampleTime_9Cycles：采样时间等于 9 个周期；
- ADC_SampleTime_16Cycles：采样时间等于 16 个周期；
- ADC_SampleTime_24Cycles：采样时间等于 24 个周期；
- ADC_SampleTime_48Cycles：采样时间等于 48 个周期；
- ADC_SampleTime_96Cycles：采样时间等于 96 个周期；
- ADC_SampleTime_192Cycles：采样时间等于 192 个周期；
- ADC_SampleTime_384Cycles：采样时间等于 384 个周期。

12）函数 ADC_SoftwareStartConv

函数功能：软件配置启动规则通道的转换。

函数原型：void ADC_SoftwareStartConv(ADC_TypeDef * ADCx)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

13）函数 ADC_GetSoftwareStartConvStatus

函数功能：获取所选的 ADC 规则转换软件启动的状态值。

函数原型：FlagStatus ADC_GetSoftwareStartConvStatus(ADC_TypeDef * ADCx)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

返回值：ADC 开始转换或没有转换，取值为 SET 或 RESET。

14）函数 ADC_EOCOnEachRegularChannelCmd

函数功能：启用或禁用每次规则通道转换完成产生 EOC。

函数原型：void ADC_EOCOnEachRegularChannelCmd(ADC_TypeDef * ADCx，FunctionalState NewState)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

输入参数 NewState：所选 ADC EOC 标志是否在每次转换完后产生，取值为 ENABLE

或 DISABLE。

15）函数 ADC_ContinuousModeCmd

函数功能：启用或禁用 ADC 连续转换模式。

函数原型：void ADC_ContinuousModeCmd(ADC_TypeDef * ADCx，FunctionalState NewState)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

输入参数 NewState：ADC 连续转换功能的状态，取值为 ENABLE 或 DISABLE。

16）函数 ADC_DiscModeChannelCountConfig

函数功能：配置所选 ADC 规则通道组的间断模式。

函数原型：void ADC_DiscModeChannelCountConfig(ADC_TypeDef * ADCx，uint8_t Number)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

输入参数 Number：规则通道间断模式的转换通道数量，取值范围为 1～8。

17）函数 ADC_DiscModeCmd

函数功能：启用或禁用规则通道组的间断模式。

函数原型：void ADC_DiscModeCmd(ADC_TypeDef * ADCx，FunctionalState NewState)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

输入参数 NewState：间断模式的启用状态，取值为 ENABLE 或 DISABLE。

18）函数 ADC_GetConversionValue

函数功能：获取规则通道的 ADC 转换结果数据。

函数原型：uint16_t ADC_GetConversionValue(ADC_TypeDef * ADCx)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

返回值：ADC 数据转换结果。

19）函数 ADC_InjectedChannelConfig

函数功能：为所选 ADC 规则通道配置其对应的规则组序号及其采样时间。

函数原型：void ADC_InjectedChannelConfig(ADC_TypeDef * ADCx，uint8_t ADC_Channel，uint8_t Rank，uint8_t ADC_SampleTime)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

输入参数 ADC_Channel：要配置的 ADC 通道，取值为 ADC_Channel_x(x=0～27)或 ADC_Channel_xb(x=0～12)。

输入参数 Rank：所要配置的通道在规则组通道的次序，取值范围为 1～4。

输入参数 ADC_SampleTime：所选择通道的采样时间，取值范围与 ADC_RegularChannelConfig 函数的 ADC_SampleTime 相同。

20）函数 ADC_InjectedSequencerLengthConfig

函数功能：用于配置注入通道组的通道数量。

函数原型：void ADC_InjectedSequencerLengthConfig(ADC_TypeDef * ADCx，uint8

_t Length)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

输入参数 Length：注入组通道数量，取值范围为 1～4。

21）函数 ADC_SetInjectedOffset

函数功能：设置注入通道转换值的偏移量。

函数原型：void ADC_SetInjectedOffset(ADC_TypeDef * ADCx, uint8_t ADC_InjectedChannel, uint16_t Offset)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

输入参数 ADC_InjectedChannel：ADC 注入通道编号，取值范围为：

- ADC_InjectedChannel_1：已选择注入的 Channel1。
- ADC_InjectedChannel_2：已选择注入的 Channel2。
- ADC_InjectedChannel_3：已选择注入的 Channel3。
- ADC_InjectedChannel_4：已选择注入的 Channel4。

输入参数 Offset：所选 ADC 注入通道的偏移值，12 位。

22）函数 ADC_ExternalTrigInjectedConvConfig

函数功能：为注入通道转换配置 ADCx 外部触发源。

函数原型：void ADC_ExternalTrigInjectedConvConfig(ADC_TypeDef * ADCx, uint32_t ADC_ExternalTrigInjecConv)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

输入参数 ADC_ExternalTrigInjecConv：指定 ADC 转换的触发源，其取值与 ADC 初始化模板中 ADC_ExternalTrigConv 的取值相同。

23）函数 ADC_ExternalTrigInjectedConvEdgeConfig

函数功能：注入通道触发的外部触发沿配置。

函数原型：void ADC_ExternalTrigInjectedConvEdgeConfig(ADC_TypeDef * ADCx, uint32_t ADC_ExternalTrigInjecConvEdge)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

输入参数 ADC_ExternalTrigInjecConvEdge：指定 ADC 转换的触发源，其取值与 ADC 初始化模板中 ADC_ExternalTrigInjecConvEdge 的取值相同。

24）函数 ADC_SoftwareStartInjectedConv

函数功能：注入通道软件启动转换设置。

函数原型：void ADC_SoftwareStartInjectedConv(ADC_TypeDef * ADCx)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

25）函数 ADC_GetSoftwareStartInjectedConvCmdStatus

函数功能：获取注入通道软件启动转换的状态。

函数原型：FlagStatus ADC_GetSoftwareStartInjectedConvCmdStatus(ADC_TypeDef * ADCx)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

返回值为 ADC 是否已启动注入通道转换,取值范围为 SET 或 RESET。

26) 函数 ADC_AutoInjectedConvCmd

函数功能:使能或禁用 ADC 在规则通道转换完后自动执行注入组转换功能。

函数原型:void ADC_AutoInjectedConvCmd(ADC_TypeDef * ADCx, FunctionalState NewState)。

输入参数 ADCx:要控制的 ADC 外设,取值为 ADC1。

输入参数 NewState:自动注入通道转换的状态,取值为 ENABLE 或 DISABLE。

27) 函数 ADC_InjectedDiscModeCmd

函数功能:为指定的 ADC 外设启用或禁用注入组通道的间断模式。

函数原型:void ADC_InjectedDiscModeCmd(ADC_TypeDef * ADCx, FunctionalState NewState)。

输入参数 ADCx:要控制的 ADC 外设,取值为 ADC1。

输入参数 NewState:注入通道间断模式的状态,取值为 ENABLE 或 DISABLE。

28) 函数 ADC_GetInjectedConversionValue

函数功能:获取指定 ADC 外设的注入通到转换结果。

函数原型:uint16_t ADC_GetInjectedConversionValue(ADC_TypeDef * ADCx, uint8_t ADC_InjectedChannel)。

输入参数 ADCx:要控制的 ADC 外设,取值为 ADC1。

输入参数 ADC_InjectedChannel:注入通道的编号,取值为:

- ADC_InjectedChannel_1:注入通道 Channel1;
- ADC_InjectedChannel_2:注入通道 Channel2;
- ADC_InjectedChannel_3:注入通道 Channel3;
- ADC_InjectedChannel_4:注入通道 Channel4。

29) 函数 ADC_ITConfig

函数功能:启用或禁用指定的 ADC 中断。

函数原型:void ADC_ITConfig(ADC_TypeDef * ADCx, uint16_t ADC_IT, FunctionalState NewState)。

输入参数 ADCx:要控制的 ADC 外设,取值为 ADC1。

输入参数 ADC_IT:指定要启用或禁用的 ADC 中断源,取值为:

- ADC_IT_EOC:转换结束中断。
- ADC_IT_AWD:模拟看门狗中断。
- ADC_IT_JEOC:注入转换结束中断。
- ADC_IT_OVR:溢出中断。

输入参数 NewState:指定 ADC 中断的状态,取值为 ENABLE 或 DISABLE。

30) 函数 ADC_GetFlagStatus

函数功能:检查指定的 ADC 标志是否设置。

函数原型:FlagStatus ADC_GetFlagStatus(ADC_TypeDef * ADCx, uint16_t ADC_

FLAG)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

输入参数 ADC_FLAG：指定要检查的标志，取值为：

- ADC_FLAG_AWD：模拟看门狗标志；
- ADC_FLAG_EOC：转换结束标志；
- ADC_FLAG_JEOC：注入组转换结束标志；
- ADC_FLAG_JSTRT：注入组转换开始标志；
- ADC_FLAG_STRT：规则组开始转换标志；
- ADC_FLAG_OVR：溢出标志；
- ADC_FLAG_ADONS：ADC ON 状态；
- ADC_FLAG_RCNR：规则通道转换未就绪；
- ADC_FLAG_JCNR：注入通道转换未就绪。

返回值：ADC_FLAG 的状态，取值为 SET 或 RESET。

31）函数 ADC_ClearFlag

函数功能：清除 ADC 标志。

函数原型：void ADC_ClearFlag(ADC_TypeDef * ADCx, uint16_t ADC_FLAG)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

输入参数 ADC_FLAG：指定要清除的标志，取值为：ADC_FLAG_AWD、ADC_FLAG_EOC、ADC_FLAG_JEOC、ADC_FLAG_JSTRT、ADC_FLAG_STRT 和 ADC_FLAG_OVR。

32）函数 ADC_GetITStatus

函数功能：检查指定的 ADC 中断是否发生。

函数原型：ITStatus ADC_GetITStatus(ADC_TypeDef * ADCx, uint16_t ADC_IT)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

输入参数 ADC_IT：指定要检查的中断源，取值为：

- ADC_IT_EOC：规则通道转换结束中断；
- ADC_IT_AWD：模拟看门狗中断；
- ADC_IT_JEOC：注入通道转换结束中断；
- ADC_IT_OVR：溢出中断。

返回值：ADC_IT 的状态，取值为 SET 或 RESET。

33）函数 ADC_ClearITPendingBit

函数功能：清除指定的 ADC 中断挂起位。

函数原型：void ADC_ClearITPendingBit(ADC_TypeDef * ADCx, uint16_t ADC_IT)。

输入参数 ADCx：要控制的 ADC 外设，取值为 ADC1。

输入参数 ADC_IT：指定要清除的中断源，取值与函数 ADC_GetITStatus 的参数 ADC_IT 相同。

11.5　ADC 案例

11.5.1　ADC 寄存器操作案例

【例 11-1】 ADC_Init 函数的寄存器操作实现。

```
void ADC_Init(ADC_TypeDef * ADCx, ADC_InitTypeDef * ADC_InitStruct)
{
  uint32_t tmpreg1 = 0;
  uint8_t tmpreg2 = 0;
  //ADCx CR1 寄存器配置
  tmpreg1 = ADCx->CR1;                                  //获取 ADCx CR1 寄存器值
  tmpreg1 &= CR1_CLEAR_MASK;                            //清除 RES 和 SCAN 域
  //根据 ADC_ScanConvMode 配置域,根据 ADC_Resolution 配置 RES 域
  tmpreg1 |= (uint32_t)(((uint32_t)ADC_InitStruct->ADC_ScanConvMode << 8) |
          ADC_InitStruct->ADC_Resolution);
  ADCx->CR1 = tmpreg1;                                  //写回到寄存器 ADCx CR1
  //配置 ADCx CR2 寄存器
  tmpreg1 = ADCx->CR2;                                  //获取 ADCx CR2 寄存器值
  //清除 CONT, ALIGN, EXTEN and EXTSEL 域
  tmpreg1 &= CR2_CLEAR_MASK;
  //根据 ADC_DataAlign 配置 ALIGN 域,根据 ADC_ExternalTrigConvEdge 配置 EXTEB 域,
  //根据 ADC_ExternalTrigConv 配置 EXTSEL 域,根据 ADC_ContinuousConvMode 配置 CONT 域
  tmpreg1 |= (uint32_t)(ADC_InitStruct->ADC_DataAlign
          | ADC_InitStruct->ADC_ExternalTrigConv
          | ADC_InitStruct->ADC_ExternalTrigConvEdge
          | ((uint32_t)ADC_InitStruct->ADC_ContinuousConvMode << 1));
  ADCx->CR2 = tmpreg1;                                  //写回寄存器 ADCx CR2
  //配置 ADCx SQR1 寄存器
  tmpreg1 = ADCx->SQR1;                                 //获取寄存器 ADCx SQR1 的值
  tmpreg1 &= SQR1_L_RESET;                              //清除 L 域
  //根据 ADC_NbrOfConversion 配置规则组通道数量 L
  tmpreg2 |= (uint8_t)(ADC_InitStruct->ADC_NbrOfConversion - (uint8_t)1);
  tmpreg1 |= ((uint32_t)tmpreg2 << 20);
  ADCx->SQR1 = tmpreg1;                                 //写回到寄存器 ADCx SQR1
}
```

【例 11-2】 ADC_Cmd 寄存器操作实现。

```
void ADC_Cmd(ADC_TypeDef * ADCx, FunctionalState NewState)
{
```

```
    if (NewState != DISABLE)
      ADCx->CR2 |= (uint32_t)ADC_CR2_ADON;                    //设置 ADON 域给 ADC 上电
    else
      ADCx->CR2 &= (uint32_t) (~ADC_CR2_ADON);                //关闭 ADC
}
```

11.5.2 ADC 库函数操作案例

1）ADC 配置流程：

- 配置 ADC 的 GPIO 为模拟输入；
- 使能 HSI 时钟，要等待 HSI 时钟开启；
- 使能 ADC 时钟；
- 配置 ADC 相关参数(转换精度，转换模式，字节对齐)；
- 配置 ADC 通道和采样时钟；
- 配置 ADC 采样频率(预分频参数)；
- 配置 ADC 中断向量相关参数；
- 开启 ADC 的 EOC 中断；
- 给 ADC 上电，并检测 ADC 是否准备好；
- 软件开启 ADC。

2）ADC 中断处理：

- 判断 EOC 中断标志位；
- 对 EOC 中断清零；
- 对转换数值处理；
- ADC 上电；
- 检测 ADONS 标志位，等待 ADC 准备好；
- 开启软件打开方式转换。

【例 11-3】 ADC 初始化配置。

```
void ADC_InitConfiguration(void)
{
  ADC_InitTypeDef ADC_InitStructure;
  RCC_HSICmd(ENABLE);
  while (RCC_GetFlagStatus(RCC_FLAG_HSIRDY) ==RESET);
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);
  RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA, ENABLE);
  GPIO_InitStructure.GPIO_Speed=GPIO_Speed_40MHz;
  GPIO_InitStructure.GPIO_Pin =GPIO_Pin_5;
  GPIO_InitStructure.GPIO_Mode =GPIO_Mode_AN;
  GPIO_Init(GPIOA, &GPIO_InitStructure);
```

```
    ADC_DeInit(ADC1);
    ADC_StructInit(&ADC_InitStructure);
    ADC_InitStructure.ADC_Resolution = ADC_Resolution_12b;
    ADC_InitStructure.ADC_ScanConvMode = DISABLE;
    ADC_InitStructure.ADC_ContinuousConvMode = ENABLE;
    ADC_InitStructure.ADC_ExternalTrigConvEdge = ADC_ExternalTrigConvEdge_None;
    ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
    ADC_InitStructure.ADC_NbrOfConversion = 1;
    ADC_Init(ADC1, &ADC_InitStructure);
    ADC_DelaySelectionConfig(ADC1, ADC_DelayLength_Freeze);
    ADC_PowerDownCmd(ADC1, ADC_PowerDown_Idle_Delay, ENABLE);
    ADC_RegularChannelConfig(ADC1, ADC_Channel_5, 1, ADC_SampleTime_96Cycles );

    //ADC_ITConfig(ADC1,ADC_IT_EOC,ENABLE);                    //中断使能
    ADC_Cmd(ADC1, ENABLE);
    while (ADC_GetFlagStatus(ADC1, ADC_FLAG_ADONS) == RESET);
}
```

启用 ADC 转换，读取数据的代码如下：

```
ADC_SoftwareStartConv(ADC1);
while (ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC) == RESET) ;
uint16_t  ADCdata = ADC_GetConversionValue(ADC1);
```

Chapter 12

第12章　低功耗技术

【导读】 低功耗时 STM32L 系列处理器的优势，在物联网电池供电系统中对于处理器的功耗要求较高。本章首先介绍了处理器的动态功耗和静态功耗，然后针对 STM32L1xx 系列处理器的电压管理、时钟管理、低功耗模式进行了介绍，并对 PWR 控制器的寄存器、库函数及典型使用方法进行了介绍。

12.1　处理器功耗的构成/类型

电源对电子设备的重要性不言而喻，它是保证系统稳定运行的基础，在很多应用场合中都对电子设备的功耗要求非常苛刻，如某些传感器信息采集设备，仅靠小型的电池提供电源，要求工作长达数年之久，且期间不需要任何维护。由于智慧穿戴设备的小型化要求，电池体积不能太大导致容量也比较小，所以也很有必要从控制功耗入手，提高设备的续行时间。STM32L 系列针对低功耗处理进行了优化，有专门的电源管理和低功耗运行模式。

功耗的构成主要有动态功耗、静态功耗、浪涌功耗这三种。

12.1.1　动态功耗

动态功耗主要是外围电路的器件功耗，包括：开关功耗（翻转功耗）和短路功耗（内部功耗）。

1. 开关功耗

开关功耗指的是数字 CMOS 电路中，对负载电容进行充放电时消耗的功耗，比如对于图 12-1 的 CMOS 非门中，当 $V_{in}=0$ 时，上面的 PMOS 导通，下面的 NMOS 截止；V_{DD} 对负载电容 C_{load} 进行充电，充电完成后，V_{out} 的电平为高电平。

当 $V_{in}=1$ 时，上面的 PMOS 截止，下面的 NMOS 导通，负载电容通过 NMOS 进行放电，放电完成后，V_{out} 的电平为低电平。充放电形成了开关功耗，开关功耗 P_{switch} 的计算公式为：

$$P_{switch}=\frac{1}{2}V_{DD}^{2}C_{load}T_{r}$$

其中，V_{DD}为供电电压，C_{load}为后级电路等效的电容负载大小，T_r 为输入信号的翻转频率。

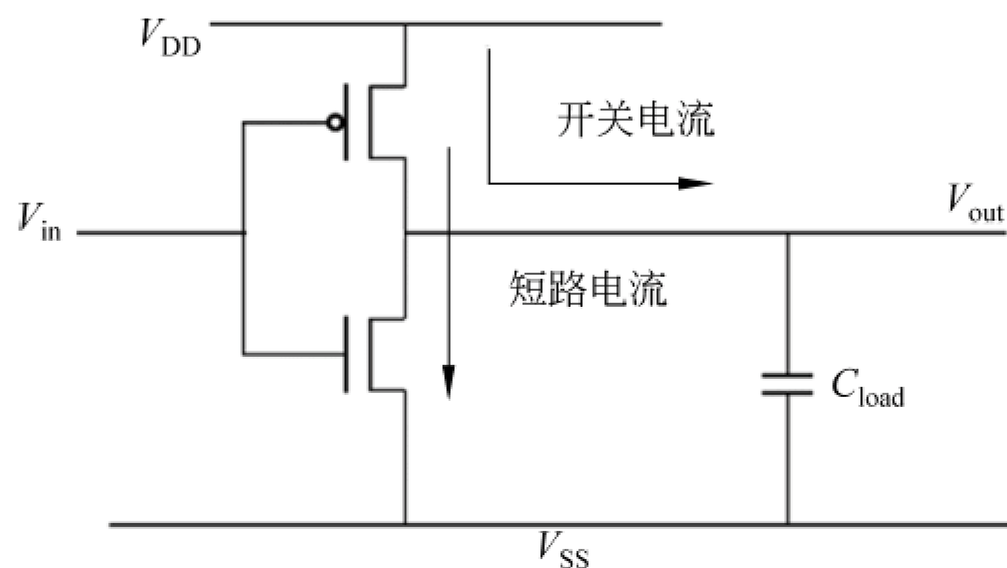


图 12-1　CMOS 非门

2. 短路功耗

短路功耗也称为内部功耗，在输入信号进行翻转时，信号的翻转不可能瞬时完成，因此 PMOS 和 NMOS 不可能总是一个截止另外一个导通，会存在一段时间，PMOS 和 NMOS 同时导通，从而导致电源 VDD 到地 VSS 之间就有短路电流，如图 12-2 的反向器电路所示。

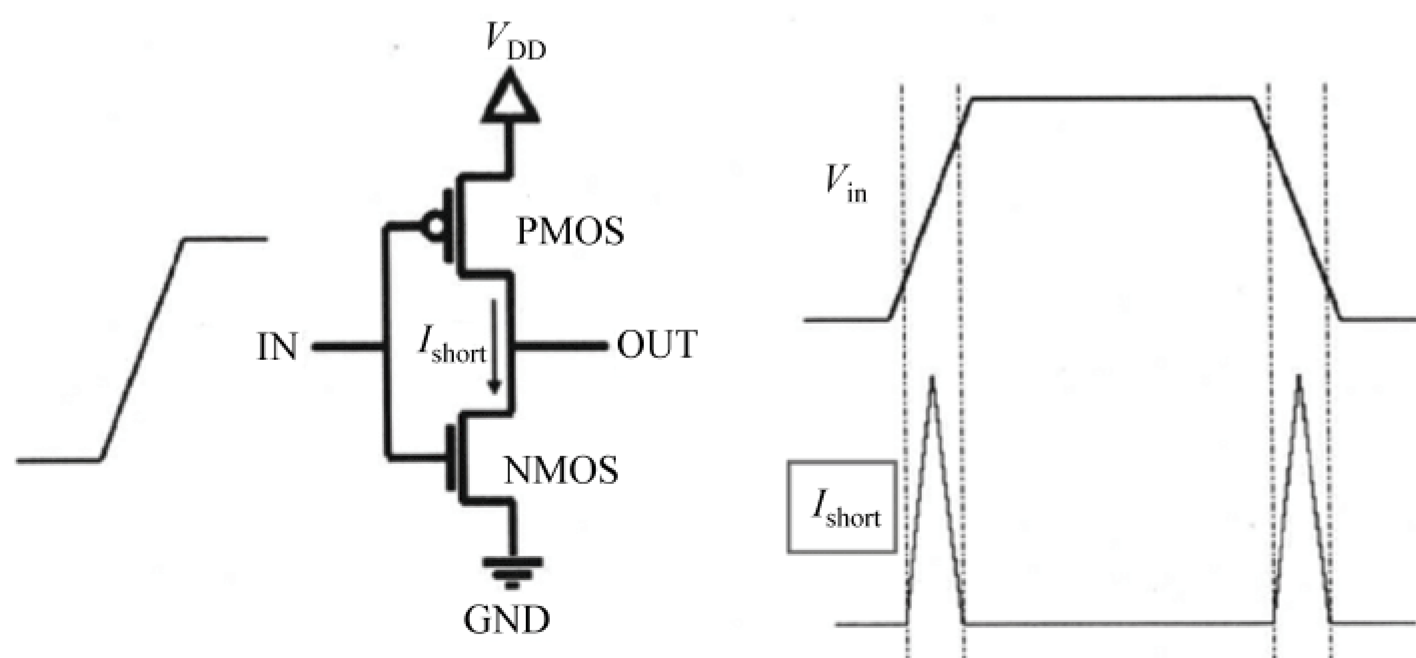


图 12-2　反相器电路的短路电流

短路功耗 P_{short}的计算公式为：$P_{short}=V_{DD}T_rQ_x$。其中，V_{DD}为供电电压，T_r 为翻转频率，Q_x 为一次翻转过程中从电源流到地的电荷量。

动态功耗主要跟电源的供电电压、翻转频率和负载电容有关。

12.1.2　静态功耗

静态功耗主要是漏电流引起的功耗，CMOS 电路漏电流下图 12-3 所示，漏电流有下面几个部分组成：

- PN 结反向电流 I_1；
- 源极和漏极之间的亚阈值漏电流 I_2；
- 栅极漏电流，包括栅极和漏极之间的感应漏电流 I_3；

- 栅极和衬底之间的隧道漏电流 I_4。

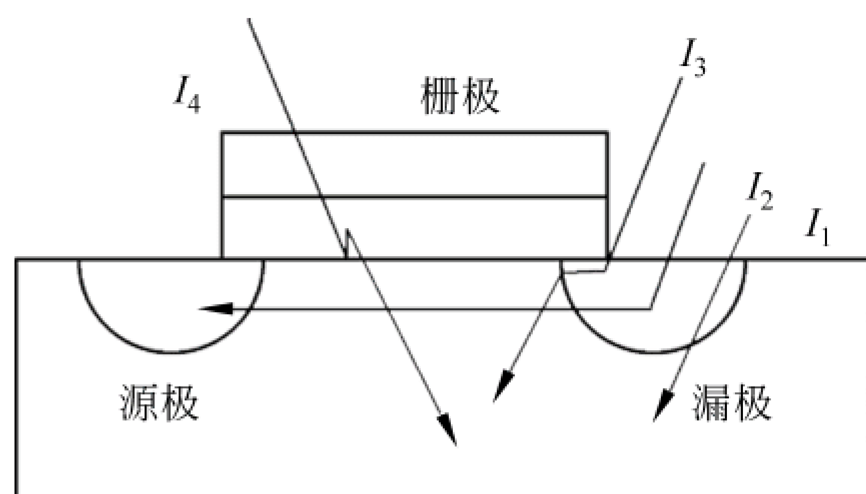


图 12-3 漏电流示意

静态功耗往往与工艺有关，静态功耗的计算公式为 $P_{peak}=V_{DD}I_{peak}$，其中，I_{peak} 为泄漏电流。

处理器的总功耗中，动态功耗特别是开关功耗占据了约 80% 的来源，因此对于低功耗设计，主要示降低开关功耗，即调整处理器工作电压、工作频率。

12.2 STM32L1 系列处理器低功耗设计

12.2.1 STM32 的电源系统

STM32L1 系列处理器支持的 VDD 供电电压范围为 1.8～3.6V，如果不支持 BOR (brown out reset) 欠压复位，则要求 VDD 供电范围为 1.65～3.6V。上节的功耗分析我们得知，电压越小，开关功耗越小，因此在低功耗模式下，尽量要使用低电压。

为了方便进行电源管理，STM32L1 系列把它的外设、内核等模块根据功能划分了供电区域，如图 12-4 所示，并集成了一个线性稳压器为内部提供 1.2～1.8V 的电压，供电模块包括：

1) V_{DD}

V_{DD} 为外部 I/O 和内部稳压器提供电压，如果 BOR 无效，V_{DD} 电压范围为 1.65～3.6V。

2) Vcore

Vcore=1.2～1.8V，Vcore 为数字外围设备、SRAM 和 Flash 提供电压。由内部稳压器产生，根据不同功耗状态，Vcore 范围是可选的(与 V_{DD} 相关)。

3) V_{DDA}

V_{DDA} 为外围模拟设备 ADC、DAC、复位模块、RC 振荡器和 PLL 提供电压，如果 BOR 无效，V_{DDA} 电压范围为 1.65～3.6V。使用 ADC 时，V_{DDA} 电压不能小于 1.8V。

4) $V_{REF}-$，$V_{ERF}+$

$V_{REF}+$ 为输入参考电压。只有在 LQFP144，UFBGA132，LQFP100，UFBGA100 和 TFBGA64 封装的才是有效的，其他封装情况下，默认连接到 V_{SSA} 和 V_{DDA} 上。

5) V_{LCD}

V_{LCD}=2.5～3.6V。LCD 控制器可由 V_{LCD}外部接口或者内部嵌入式升压转换器提供电压。

STM32L1 系列处理器的电源分配特点为:

1) 独立的 AD 和 DAC 转换器供给和参考电压

为了提高转换精度,ADC 和 DAC 有独立的电压供给,可以对数字电源滤波和隔离提供单独的 V_{DDA}和 V_{SSA}为 ADC 提供电压。

2) 独立的 LCD 供给电压

V_{LCD}可用于控制 LCD 的对比度。该电源可以采用外部电路供电,要求电压范围应为 2.56～3.6V,可以与 V_{DD}无关,也可以连接到一个外部电容上,用于 MCU 的升压转换器软件控制 LCD 的电压。

3) 电压调整器

线性电压调整器可用于所有数字电路(除了待机中的电路)。调整器的输出电压(Vcore)1.2～1.8V,可以通过编程设置为三种不同的范围。复位后,电压调整器被使能。可以配置为三种模式:主电压调节器模式 MR,低功耗运行 LPR 和待机模式。

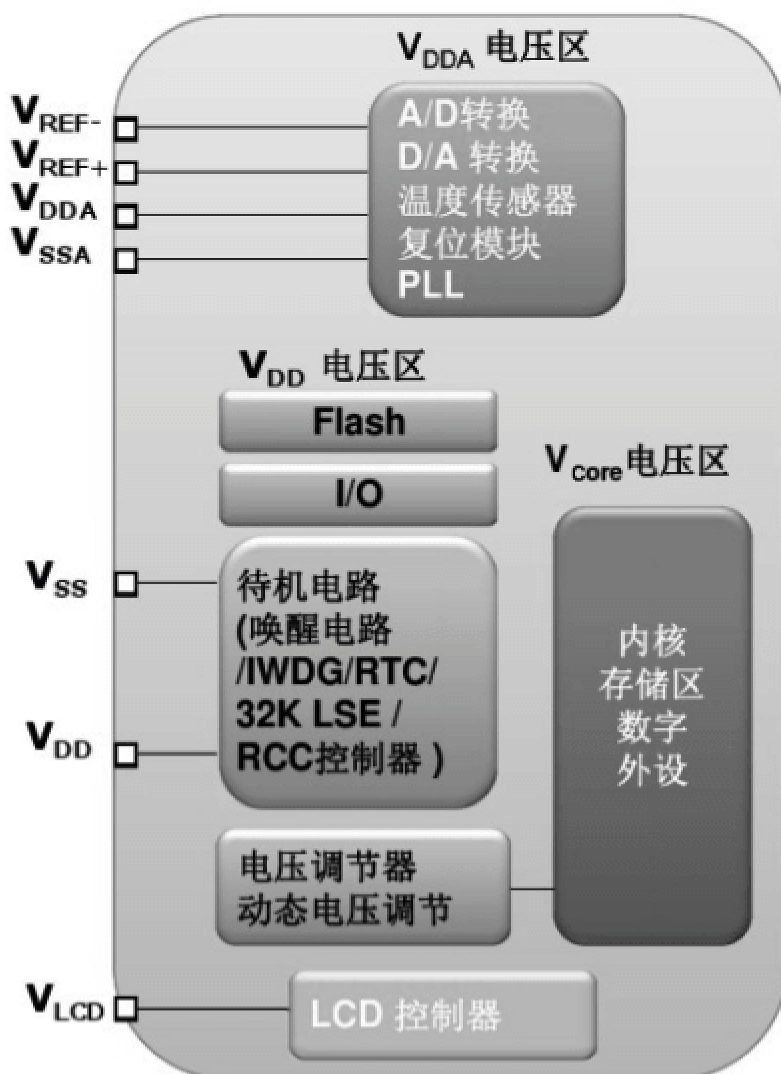


图 12-4 电源分配

- 在运行模式下,调整器处于主模式(MR),并为 Vcore 提供全功率供电(处理器核心、存储器、数字外设);
- 在低功耗运行模式下,调整器处于低功耗模式(LWR),并为 Vcore 提供供电,维持寄存器及内部 SRAM 数据;
- 在睡眠模式下,调整器处于主模式(MR),并为 Vcore domain 提供供电,维持寄存器及内部 SRAM 的数据;
- 在低功耗睡眠模式下,调整器处于低功耗模式(LWR),并未 Vcore 提供供电,维持寄存器及内部 SRAM 数据;
- 在待机模式下,调整器处于关闭状态,除了待机电路之后,寄存器及 SRAM 中的数据都会丢失。

12.2.2 动态电压调节管理

动态电压调节管理是一项电源管理技术,根据不同的情况,增加或减少 Vcore 的电压。提高设备的性能或降低设备功耗。

根据应用情况可将处理器的电压范围分为三种:Range1、Range2 和 Range3。

(1) Range1 为高性能范围。电压调节器输出 1.8V 电压(V_{DD}电压为 2.0V)。在该范围

内，Flash 编程和擦除操作均可操作。

（2）Range2 为中等性能范围，电压调节器输出 1.5V，Flash 存储器有效，但读取时间为中等，可以编程和擦除 Flash。

（3）Range3 为低性能范围，电压调节器输出 1.2V，Flash 存储器有效，但读取时间较慢，不可以编程和擦除 Flash。

三种区间的性能如表 12-1 所示，CPU 性能、V_{DD}与 Vcore 之间的关系如图 12-5 所示。

表 12-1 不同电压范围的性能

CPU 性能	功耗	电压范围	电压值/V	最高频率/MHz	V_{DD}
高	高	1	1.8	32	2.0～3.6
中	中	2	1.5	16	1.65～3.6
低	低	3	1.2	4	1.65～3.6

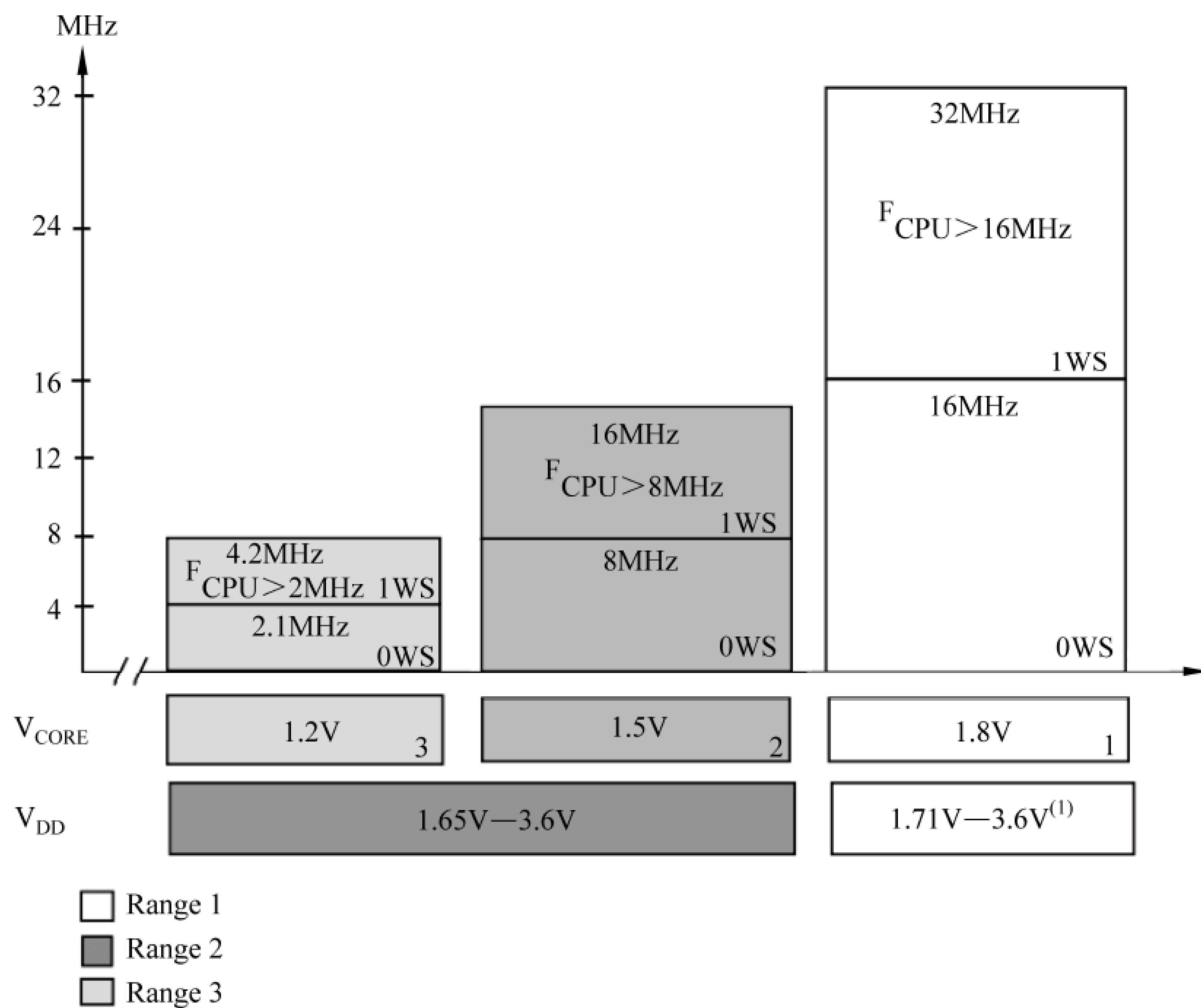


图 12-5 V_{DD}、Vcore 和处理器性能的关系

动态电压调节配置的流程如下：

- 检测 V_{DD}以确认可使用哪些 ranges；
- 轮询 PWR_CSR 寄存器的 VOSF 位直到该位为 0；
- 通过配置 PWR_CR 寄存器的 VOS[12：11]位来配置电压范围；
- 轮询 VOSF 位直到该位为 0。

12.2.3 电源检测

STM32L1xx 内部集成了上电复位(POR)/掉电复位(PDR)、欠压复位(BOR)电路。当设备工作在 1.8～3.6V 时,BOR 默认情况下是使能的,当 V_{DD}电压降低到了 1.8V 的阈值时,将引起复位。可以通过修改阈值或者关闭 BOR 使得 V_{DD}的最小值为 1.65V。

BOR 的 5 种阈值可以通过选择字节进行配置。为了减少待机模式下的功耗,内部电压参考 VERFINT 可以自动关闭。当 V_{DD}低于指定的阈值时(VPOR,VPDR,VBOR),设备保持在复位模式,并且无需任何外部复位电路。

STM32L1xx 内嵌可编程的电压检测器,用于监测 V_{DD}/V_{DDA}的电压和跟 VPVD 阈值进行比较。可以选择 7 种不同的 PVD。当 V_{DD}/V_{DDA}低于或者高于 VPVD 阈值的时候会产生一个中断,中断可以例行的可以产生警告信息或者令 MCU 进入到安全的状态中,需要通过应用来使能 PVD。

1) 上电复位(POR)/掉电复位(PDR)

当上电时,V_{DD}/V_{DDA}低于一个特定的阈值 VPOR 时,设备保持在复位状态,且不需要多余的外部复位电路。POR 默认是使能的,默认阈值为 1.5V。V_{DD}下降到低于 VPDR 阈值时,PDR 让设备保持在复位状态。PDR 也是默认使能的,阈值为 1.5V。只有当 BOR 是无效时才可使用 POR 和 PDR。

2) 欠压复位(BOR)

当设备工作在 1.65～3.6V 时,BOR 无效,电压检测用 POR/PDR。当设备工作在 1.8～3.6V 时,BOR 在上电后使能,且其阈值 VBOR 为 1.8V。VBOR 可以通过选项字节进行修改,默认情况下为 Level 4:

- BOR Level 0(VBOR0):复位阈值为:1.69～1.80 V。
- BOR Level 1(VBOR1):复位阈值为:1.94～2.1 V。
- BOR Level 2(VBOR2):复位阈值为:2.3～2.49 V。
- BOR Level 3(VBOR3):复位阈值为:2.54～2.74 V。
- BOR Level 4(VBOR4):复位阈值为:2.77～3.0 V。

当 V_{DD}下降到低于所选择的 VBOR 阈值,则会产生复位。当 V_{DD}上升到高于 VBOR 上限时,则会释放复位,设备启动。

3) 可编程电压检测器(PVD)

可以使用 PVD 来检测 VDD 电源,并与 PWR_CR 寄存器 PLS[2:0]定义的阈值进行比较,通过 PWR_CSR 寄存器的 PVODO 标志识别 VDD 是低于还是高于 PVD 阈值。该事件连接到了外部中断控制器的 EXTI16,若配置了 EXTI 寄存器则可产生中断。当 V_{DD}上升到高于 PVD 阈值或者下降到低于 PVD 阈值时(与 EXTI16 上的边缘配置有关)会产生中断。可应用于通过中断服务处理紧急的关机任务。

4) 内部参考电压(VERFINT)

内部参考电压的功耗并不是可忽略的,为了减少功耗,可以通过 PWR_CR 寄存器的

ULP 位令内部参考电压失效。

12.2.4 低功耗模式

系统启动后默认处于运行模式，CPU 时钟由 HCLK 驱动。当 CPU 不需要保持在运行模式下时，可以进入多种低功耗模式。STM32L1 系列提供了五种低功耗模式：

- 低功耗运行模式：电压调节器处于低功耗模式，限制了时钟频率和可运行外设数量；
- 睡眠模式：Cortex-M3 停止，外设保持运行；
- 低功耗睡眠模式：Cortex-M3 停止，限制了时钟频率和可运行外设数量，电压调节器处于低功耗模式，RAM 掉电，Flash 停止；
- 停止模式：所有时钟停止，电压调节器继续运行，并处于低功耗模式；
- 待机模式：Vcore 掉电关机。

另外，可以通过减少系统时钟频率、关闭不使用的外设时钟减少运行时功耗。

各种低功耗模式的进入和换序条件如图 12-6 所示，不同低功耗模式的电流消耗如图 12-7 所示。

模式	进入	唤醒	对VCORE域时钟影响	对VDD域时钟影响	内部电源变换器	I/O 状态	唤醒延迟
低功耗运行模式	设置LPSDSR 位和LPRUN 位 + 设置系统时钟	恢复系统时钟和内部电源变换器的工作模式	无	无	低功耗模式	所有I/O口的状态和运行时保持一致	无
睡眠模式	WFI	任意中断	CPU时钟关闭，不影响其他时钟	无	正常模式		无
	WFE	唤醒事件					
低功耗睡眠模式	设置LPSDSR 位 + WFI	任意中断		无	低功耗模式		电源变换器的改变时间+FLASH的唤醒时间
	设置LPSDSR 位 + WFE	唤醒事件					
停止模式	设置PDDS和LPSDSR位 + 设置SLEEPDEEP位 + WFI 或 WFE	任意EXTI中断 (在EXTI寄存器中配置的**内部**或者**外部**中断源)	所有的VCORE域时钟都关闭	HSI，HSE和MSI都关闭	正常模式/低功耗模式		MSI RC的唤醒时间+电源变换器的唤醒时间 + FLASH的唤醒时间 唤醒的典型值为7.9us
待机模式	设置PDDS和 + 设置SLEEPDEEP位 + WFI 或 WFE	WKUP引脚的上升沿，RTC报警 (Alarm A 或者Alarm B)，RTC唤醒事件，RTC时间戳事件，RTC侵入事件，NRST引脚的外部复位，IWDG复位			关闭	所有I/O口都保持在高阻状态	V_{REFINT} 开的情况下唤醒典型值为57.2us V_{REFINT} 关的情况下唤醒典型值为2.4ms

图 12-6 不同模式的进入和唤醒条件

1. 低功耗模式下的时钟

1）睡眠和低功耗睡眠模式

在睡眠和低功耗睡眠模式下，CPU 的时钟处于停止状态。寄存器接口时钟和所有外设

模式	STM32L15x典型值	STM32F10x 典型值
运行模式功耗 代码在Flash中运行，内核供电范围选择3，开外设时钟	230μA/MHz	-
运行模式功耗 代码在RAM中运行，内核供电范围选择3，开外设时钟	186μA/MHz	-
低功耗运行模式功耗 代码在RAM中运行，使用内部RC(32kHz的MSI)，开外设时钟	10.4μA	-
睡眠模式功耗 代码在Flash中运行，主时钟频率为16 MHz，关所有外设时钟	650μA	-
睡眠模式功耗 代码在Flash中运行，主时钟频率为16 MHz，开所有外设时钟	2.5mA	-
低功耗睡眠模式功耗 代码在Flash中运行，主时钟频率为32kHz, 内部电源变换器工作在低功耗模式下，运行一个32kHz的定时器	6.1μA	-
停止模式功耗 内部电源变换器工作在低功耗模式下，关闭低速/高速内部振荡器和高速外部振荡器，不使能独立看门狗	0.43μA w/o RTC 1.3μA w/ RTC	14 μA
待机模式功耗 使用低速内部振荡器，不使能独立看门狗，关闭RTC	0.27μA	2 μA
待机模式功耗 使能RTC	1.0 μA	-

图 12-7　不同模式的电流消耗及其和 F10x 系列的对比

时钟均可通过软件进行关闭。当处于低功耗睡眠模式时，RAM 接口时钟处于掉电状态。AHB 总线到 APB 总线的桥时钟可以通过硬件进行关闭。

2）停止和待机模式

在停止和待机模式下，系统时钟和所有高速时钟均是处于停止状态：

- PLL 无效；
- 内部 RC 16MHz(HSI)振荡器无效；
- 外部 1M～24MHz(HSE)振荡器无效；
- 内部 65kHZ～4MHz(MSI)振荡器无效。

当通过中断退出停止模式或者复位退出待机模式时，内部 MSI 被选择为系统时钟。当设备退出停止模式时，之前的 MSI 配置仍是有效的。当设备退出待机模式时，被重置为默认的 2MHZ。

运行模式下，可以通过降低系统时钟和关闭外设时钟降低功耗。外设时钟受以下寄存器 RCC_AHBENR、RCC_APB2ENR、RCC_APB1ENR 控制，在睡眠模式时，为了关闭外设时钟可以通过复位 RCC_AHBLPENR 和 RCC_APBxLPENR 相应的位。

2. 低功耗运行模式

在运行时，为了进一步减少功耗，电压调整器可以配置为低功耗模式，在该模式下，系统的频率不应超过 f_MSI Range1。

1）进入低功耗模式

- 通过 RCC_APBxENR 和 RCC_AHBENR 寄存器使能或者关闭每一个数字 IP

时钟；

- 系统时钟频率需要下降到不得超过 f_MSI Range1；
- 通过置位 LPRUN 和 LPSDSR 强制让调整器运行在低功耗模式下。

2）退出低功耗模式

- 配置调节器运行在主电压调节器模式；
- 如果有需要，开启 Flash 存储器；
- 按需要增加系统时钟频率。

3. 睡眠模式

1）进入睡眠模式

通过执行 WFI 或 WEF 可以进行到睡眠模式下，根据 Cortex-M3 SCR 寄存器的 SLEEPONEXIT 位，睡眠模式的进入机制有两种：

- 立即睡眠：如果 SLEEPONEXIT 被清除，当 WFI/WEF 指令被执行时，MCU 则进入睡眠模式；
- 异常退出睡眠：如果 SLEEPONEXIT 被置位，当最低优先级的 ISR 退出时，MCU 则进入睡眠模式。

2）退出睡眠模式

- 如果是通过 WFI 进入到睡眠模式的，发生任何能被 NVIC 所确认的外设中断均会将设备从睡眠模式中唤醒；
- 如果是通过 WEF 进入到睡眠模式的，一检测到有事件发生时，设备将从睡眠模式中唤醒。

4. 低功耗睡眠模式

1）进入低功耗睡眠模式

通过配置电压调整器处于低功耗模式，并且执行 WFI/WEF 指令，则可进入到低功耗睡眠模式。在该模式下，Flash 不可用，RAM 存储器可用。在该模式下，系统时钟频道不应高于 f_MSI Range1。只有当 Vcore 处于 Range2 时，才可进入低功耗睡眠模式。

低功耗睡眠同样有立即睡眠和异常退出睡眠两种模式，进入睡眠的流程为：

- 可通过控制位关闭 Flash，减少功耗；
- 通过 RCC_APBxENR 和 RCC_AHBENR 寄存器使能或者关闭每一个数字外设时钟；
- 必须减少系统时钟频率；
- 强制电压调节器进入到低功耗模式（LPSDSR 位）；
- 执行 WFI/WEF 指令以进入到睡眠模式。

2）退出低功耗睡眠模式

- 如果是通过 WFI 进入到睡眠模式的，发生任何能被 NVIC 所确认的外设中断均会将设备从睡眠模式中唤醒；
- 如果是通过 WEF 进入到睡眠模式的，一检测到有事件发生时，设备将从睡眠模式中唤醒。

5. 停止模式

停止模式基于 Cortex-M3 深度睡眠模式，电压调节器可以配置为正常或者低功耗模式。

在停止模式时，所有位于 Vcore 的时钟停止，PLL、MSI、HSI 和 HSE 均无效，内部 SRAM 和寄存器将被保留。为了在停止模式下获取更低功耗，内部 Flash 一般配置为低功耗模式，在进入停止模式前，可关闭 VREFINT，BOR，PVD 和温度传感器。在退出停止模式时通过软件设置使之前关闭的功能重新开启。

为了进一步减少停止模式下的功耗，通过配置 PWR_CR 寄存器的 LPSDSR 位内部电压调节器可以设置为低功耗模式，停止模式下，以下功能模块可以进行单独配置：

- 独立看门狗(IWDG)：写其主要的寄存器或者硬件选项来启动 IWDG；
- 实时时钟(RTC)：配置 RCC_CSR 寄存器的 RTCEN 位；
- 内部 RC 振荡器(LSI RC)：配置 RCC_CSR 寄存器的 LSION 位；
- 外部 32.768kHZ 振荡器(LSE OSC)：配置 RCC_CSR 寄存器的 LSEON 位。

ADC，DAC 或 LCD 在停止模式也会消耗能源，最好将 ADC_CR2 寄存器的 ADON 和 DAC_CR 寄存器的 ENx 位需要配置为 0。

当退出停止模式时会产生一个中断或者唤醒事件，MSI RC 振荡器被选择为系统时钟。

6. 待机模式

待机模式下会得到最低的功耗，电压调节器关闭，Vcore 关闭，PLL、MSI、HSI 和 HSE 失效。除了 RTC 寄存器、RTC 备份寄存器，待机电路外 SRAM 和寄存器上下文均被丢失。

12.3 功耗控制寄存器

1. PWR 功耗控制寄存器 PWR_CR

PWR 功耗控制寄存器的有效域定义如图 12-8 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	LPRUN	Res.	VOS[1:0]		FWU	ULP	DBP	PLS[2:0]			PVDE	CSBF	CWUF	PDDS	LPSDSR
	rw		rw	rw	rw	rw	rw	rw	rw	rw	rw	rc_w1	rc_w1	rw	rw

图 12-8 PWR 功耗控制寄存器

LPRUN：低功耗运行模式，当 LPRUN 位与 LPSDSR 位一起置 1 时，电压调节器从主模式切换到低功耗模式。否则，它仍处于主模式。0 表示主电压电压调节器模式，1 表示电压调节器低功耗模式。

VOS[1：0]：电压调整范围选择，00 表示禁止改变电压范围，01～11 分别表示 Range1～

Range3。

FWU、ULP：快速唤醒和超低功耗模式，该位与 ULP 位配合使用。如果 ULP=0，则忽略 FWU；如果 ULP=1 且 FWU=1，则从低功耗模式退出时，将忽略 VREFINT 启动时间；如果 ULP=1 且 FWU=0，则仅在 VREFINT 准备就绪时退出低功耗模式。

DBP：禁用备份写保护，0 表示复位时禁止访问 RTC，RTC 备份和 RCC CSR 寄存器，1 表示允许。

PLS[2：0]：PVD 电平选择，000～110 分别表示 1.9V、2.1V、2.3V、2.5V、2.7V、2.9V、3.1V，111 表示 PVD 为外部输入模拟电压（内部与 VREFINT 比较），此时 PVD_IN 输入（PB7）必须配置为模拟输入。

PVDE：电源电压检测器使能，0 表示 PVD 禁用，1 表示启用。

CSBF：清除待机标志，写 1 表示清除 SBF 待机标志。

2. PWR 功耗控制/状态寄存器 PWR_CSR

PWR 功耗控制/状态寄存器的有效域定义如图 12-9 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved					EWUP3	EWUP2	EWUP1	Reserved		REGLPF	VOSF	VREFINTRDYF	PVDO	SBF	WUF
					rw	rw	rw			r	r	r	r	r	r

图 12-9　PWR 功耗控制/状态寄存器

EWUP3～EWUP1：使能 WKUP 引脚 3～1，0 表示 WKUP 引脚 3～1 用于通用 I/O，1 表示 WKUP 引脚 3～1 用于从待机模式唤醒，强制输入下拉，WKUP 引脚 3～1 的上升沿将系统从待机模式唤醒。

REGLPF：调节器 LP 标志，当 MCU 处于低功耗运行模式时，该位由硬件置 1，当 MCU 退出低功耗运行模式时，该位保持为 1 直到电压调节器进入主模式。0 表示电压调节器在主模式下准备就绪，1 表示电压调节器处于低功耗模式。

VOSF：电压调节选择标志，在更改电压范围后，内部稳压器需要一定的时间，VOSF 位表示电压调节器已达到 VOS 位定义的电压电平，1 表示未达到。

VREFINTRDYF：内部参考电压（VREFINT）就绪标志位，0 表示 VREFINT 关闭，1 表示 VREFINT 准备就绪。

PVDO：PVD 输出，该位由硬件置 1 和清除。仅当 PVDE 位使能 PVD 时有效，0 表示 VDD 高于 PLS[2：0]位选择的 PVD 阈值，1 表示低于 PVD 阈值。

SBF：待机标志，该位由硬件置 1，仅由 POR/PDR 清除，或者通过设置 PWR 功率控制寄存器（PWR_CR）中的 CSBF 位置 1，0 表示设备未处于待机模式，1 表示处于待机模式。

WUF：唤醒标志，该位由硬件置 1，并由系统复位或通过设置 CWUF 位清零，0 表示没有发生唤醒事件，1 表示收到唤醒事件。

12.4　PWR 寄存器结构及库函数

12.4.1　PWR 寄存器结构

PWR 寄存器结构，PWR_TypeDeff 在文件 stm32l1xx.h 中定义如下：

```
typedef struct
{
  __IO uint32_t CR;
  __IO uint32_t CSR;
} PWR_TypeDef;
```

ADC 外设声明于文件 stm32l1xx.h：

```
#define PERIPH_BASE                ((uint32_t)0x40000000)
#define APB1PERIPH_BASE PERIPH_BASE
#define APB2PERIPH_BASE            (PERIPH_BASE + 0x10000)
#define AHBPERIPH_BASE             (PERIPH_BASE + 0x20000)
#define PWR_BASE                   (APB1PERIPH_BASE + 0x7000)
#define PWR                        ((PWR_TypeDef * ) PWR_BASE)
```

12.4.2　PWR 库函数

PWR 库函数如表 12-2 所示。

表 12-2　PWR 库函数

函　　数	功　　能
PWR_DeInit	PWR 寄存器复位
PWR_RTCAccessCmd	RTC 备份写保护启用或禁用
PWR_PVDLevelConfig	PVD 参考电压设置
PWR_PVDCmd	PVD 启用或禁用
PWR_WakeUpPinCmd	启用或禁用唤醒引脚功能
PWR_FastWakeUpCmd	低功耗模式快速唤醒使能
PWR_UltraLowPowerCmd	启用或禁用内部参考电源低功耗模式
PWR_VoltageScalingConfig	设置 Vcore 电压范围
PWR_EnterLowPowerRunMode	进入低功耗运行模式

续表

函　　数	功　　能
PWR_EnterSleepMode	进入休眠模式
PWR_EnterSTOPMode	进入停止模式
PWR_EnterSTANDBYMode	进入待机模式
PWR_GetFlagStatus	获取状态寄存器
PWR_ClearFlag	清除状态位

1）函数 PWR_RTCAccessCmd

函数功能：启用或禁用对 RTC 和备份寄存器的访问。

函数原型：void PWR_RTCAccessCmd(FunctionalState NewState)。

输入参数 NewState：是否允许访问，取值范围为 ENABLE 或 DISABLE。

2）函数 PWR_RTCAccessCmd

函数功能：配置电源电压检测器(PVD)的电压阈值。

函数原型：void PWR_PVDLevelConfig(uint32_t PWR_PVDLevel)。

输入参数 PWR_PVDLevel：PVD 电压值，取值范围为：

- PWR_PVDLevel_0：PVD 检测电平设置为 1.9V。
- PWR_PVDLevel_1：PVD 检测电平设置为 2.1V。
- PWR_PVDLevel_2：PVD 检测电平设置为 2.3V。
- PWR_PVDLevel_3：PVD 检测电平设置为 2.5V。
- PWR_PVDLevel_4：PVD 检测电平设置为 2.7V。
- PWR_PVDLevel_5：PVD 检测电平设置为 2.9V。
- PWR_PVDLevel_6：PVD 检测电平设置为 3.1V。
- PWR_PVDLevel_7：外部输入模拟电压(内部比较)。

3）函数 PWR_PVDCmd

函数功能：启用或禁止电源电压检测器(PVD)。

函数原型：void PWR_PVDCmd(FunctionalState NewState)。

输入参数 NewState：是否启用 PVD，取值为 ENABLE 或 DISABLE。

4）函数 PWR_WakeUpPinCmd

函数功能：启用或禁用 I/O 唤醒功能。

函数原型：void PWR_WakeUpPinCmd(uint32_t PWR_WakeUpPin，FunctionalState NewState)。

输入参数 PWR_WakeUpPin：指定要配置的 I/O 引脚，取值为 PWR_WakeUpPin_x(x=1，2,3)。

输入参数 NewState：是否启用 I/O 作为唤醒引脚，取值为 ENABLE 或 DISABLE。

5）函数 PWR_WakeUpPinCmd

函数功能：启用或禁用 I/O 唤醒功能。

函数原型：void PWR_WakeUpPinCmd(uint32_t PWR_WakeUpPin，FunctionalState NewState)。

输入参数 PWR_WakeUpPin：指定要配置的 I/O 引脚，取值为 PWR_WakeUpPin_x(x=1，2，3)。

输入参数 NewState：是否启用 I/O 作为唤醒引脚，取值为 ENABLE 或 DISABLE。

6）函数 PWR_VoltageScalingConfig

函数功能：配置电压调节器电压范围。

函数原型：void PWR_VoltageScalingConfig(uint32_t PWR_VoltageScaling)。

输入参数 PWR_VoltageScaling：指定电压范围，取值为 PWR_VoltageScaling_Rangex (x=1，2，3)。

7）函数 PWR_EnterLowPowerRunMode

函数功能：是否进入低功耗运行模式。

函数原型：void PWR_EnterLowPowerRunMode(FunctionalState NewState)。

输入参数 NewState：是否进入低功耗运行模式，取值为 ENABLE 或 DISABLE。

8）函数 PWR_EnterSleepMode

函数功能：是否进入进入睡眠模式。

函数原型：void PWR_EnterSleepMode(uint32_t PWR_Regulator，uint8_t PWR_SLEEPEntry)。

输入参数 PWR_Regulator：休眠模式下的电压调节器状态，取值为：

- PWR_Regulator_ON：电压调节器开启；
- PWR_Regulator_LowPower：电压调节器低功耗模式。

输入参数 PWR_SLEEPEntry：指定是否使用 WFI 或 WFE 指令进入 SLEEP 模式。其取值为：

- PWR_SLEEPEntry_WFI：使用 WFI 指令进入休眠模式；
- PWR_SLEEPEntry_WFE：使用 WFE 指令进入休眠模式。

9）函数 PWR_EnterSTOPMode

函数功能：是否进入进入停止模式。

函数原型：void PWR_EnterSTOPMode(uint32_t PWR_Regulator，uint8_t PWR_STOPEntry)。

输入参数 PWR_Regulator：停止模式下的电压调节器状态，取值与 PWR_EnterSleepMode 函数的参数相同。

输入参数 PWR_STOPEntry：指定是否使用 WFI 或 WFE 指令进入 STOP 模式。其取值为：

- PWR_STOPEntry_WFI：使用 WFI 指令进入休眠模式；
- PWR_STOPEntry_WFE：使用 WFE 指令进入休眠模式。

10）函数 PWR_EnterSTANDBYMode

函数功能：是否进入待机模式。

函数原型：void PWR_EnterSTANDBYMode(void)。

11）函数 PWR_GetFlagStatus

函数功能：检查是否设置了指定的 PWR 标志。

函数原型：FlagStatus PWR_GetFlagStatus(uint32_t PWR_FLAG)。

输入参数 PWR_FLAG：指定要检查的标志，取值为：

- PWR_FLAG_WU：唤醒标志；
- PWR_FLAG_SB：待机标志；
- PWR_FLAG_PVDO：PVD 输出；
- PWR_FLAG_VREFINTRDY：内部电压参考就绪标志；
- PWR_FLAG_VOS：电压调节选择标志；
- PWR_FLAG_REGLP：电压调节器低功耗模式标志。

返回值 PWR_FLAG 为状态值，取值为 SET 或 RESET。

12）函数 PWR_GetFlagStatus

函数功能：清除指定的 PWR 标志。

函数原型：void PWR_ClearFlag(uint32_t PWR_FLAG)。

输入参数 PWR_FLAG：指定要清除的标志，取值为：

- PWR_FLAG_WU：唤醒标志；
- PWR_FLAG_SB：待机标志；

12.5 PWR 案例

【例 12-1】 寄存器级操作案例。

```
PWR_EnterSTOPMode(uint32_t PWR_Regulator, uint8_t PWR_STOPEntry)
{
  uint32_t tmpreg = 0;
  //配置 STOP 模式下的电压调节器
  tmpreg = PWR->CR;
  tmpreg &= CR_DS_MASK;                    //清除 PDDS 和 LPDSR 域
  tmpreg |= PWR_Regulator;                 //根据输入的 PWR_Regulator 设置 LPDSR 域
  PWR->CR = tmpreg;                        //配置值写回到控制寄存器 CR
  /* Set bit of Cortex System Control Register */
  SCB->SCR |= SCB_SCR_SLEEPDEEP;           //配置 Cortex-M3 系统控制寄存器的 SLEEPDEEP 域
  if(PWR_STOPEntry == PWR_STOPEntry_WFI)   //选择停止模式的进入条件为 WFI
  {
    __WFI();                               //休眠,等待中断唤醒
```

```
    }
    Else                                //进入条件为 WFE
    {
        __WFE();                        //休眠,等待事件发生
    }
    //清除 Cortex-M3 系统控制寄存器的 SLEEPDEEP 域
    SCB->SCR &= (uint32_t)~((uint32_t)SCB_SCR_SLEEPDEEP);
}
```

【例 12-2】 基于库函数的低功耗运行模式配置。

```
void LowPowerRunMode_Measure(void)
{
    //系统时钟 MSI Range0 (65kHz)
    RCC_DeInit();                                   //RCC 复位
    FLASH_SetLatency(FLASH_Latency_0);              //Flash 0 等待
    FLASH_PrefetchBufferCmd(DISABLE);               //禁止预取
    FLASH_ReadAccess64Cmd(DISABLE);                 //禁止 64 位访问
    //使能 PWR APB1 时钟
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR, ENABLE);
    //选择电压调节器为 Range2 (1.5V)
    PWR_VoltageScalingConfig(PWR_VoltageScaling_Range2);
    //等待电压调节器稳定
    while(PWR_GetFlagStatus(PWR_FLAG_VOS) != RESET);
    RCC_HCLKConfig(RCC_SYSCLK_Div2);                // HCLK = SYSCLK/2 = 32kHz
    RCC_PCLK2Config(RCC_HCLK_Div1);                 // PCLK2 = HCLK
    RCC_PCLK1Config(RCC_HCLK_Div1);                 // PCLK1 = HCLK
    RCC_MSIRangeConfig(RCC_MSIRange_0);             // MSI 设置为 65.536kHz
    RCC_SYSCLKConfig(RCC_SYSCLKSource_MSI);         // MSI 作为系统时钟源
    while (RCC_GetSYSCLKSource() != 0x00) ;
    //配置所有的 GPIO 为模拟,减少功耗
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA | RCC_AHBPeriph_GPIOB |
    RCC_AHBPeriph_GPIOC | RCC_AHBPeriph_GPIOD | RCC_AHBPeriph_GPIOE |
    RCC_AHBPeriph_GPIOH | RCC_AHBPeriph_GPIOF | RCC_AHBPeriph_GPIOG, ENABLE);
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AN;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_40MHz;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_All;
    GPIO_Init(GPIOC, &GPIO_InitStructure);
    GPIO_Init(GPIOD, &GPIO_InitStructure);
    GPIO_Init(GPIOE, &GPIO_InitStructure);
    GPIO_Init(GPIOH, &GPIO_InitStructure);
    GPIO_Init(GPIOF, &GPIO_InitStructure);
    GPIO_Init(GPIOG, &GPIO_InitStructure);
```

```
    GPIO_Init(GPIOA, &GPIO_InitStructure);
    GPIO_Init(GPIOB, &GPIO_InitStructure);
    //关闭 GPIO时钟

    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA | RCC_AHBPeriph_GPIOB |
    RCC_AHBPeriph_GPIOC | RCC_AHBPeriph_GPIOD | RCC_AHBPeriph_GPIOE |
    RCC_AHBPeriph_GPIOH | RCC_AHBPeriph_GPIOF | RCC_AHBPeriph_GPIOG, DISABLE);
    //进入低功耗运行模式
    PWR_EnterLowPowerRunMode(ENABLE);
    while(PWR_GetFlagStatus(PWR_FLAG_REGLP) ==RESET);
    //此时进入 LP模式
    //退出 LP RUN 模式
    PWR_EnterLowPowerRunMode(DISABLE);
    while(PWR_GetFlagStatus(PWR_FLAG_REGLP) !=RESET);
}
```

参 考 文 献

[1] Joseph Yiu. Cortex-M3 权威指南[M]. 宋岩,译. 北京:北京航空航天大学出版社,2009.

[2] 陈泽宇. 计算机组成与系统结构[M]. 北京:清华大学出版社,2009.

[3] Stmicroelectronics. RM0038: STM32L100xx, STM32L151xx, STM32L152xx and STM32L162xx advanced ARM-based 32-bit MCUs[EB/OL]. (2017-09-04)[2018-02-01]. https://www.st.com/content/ccc/resource/technical/document/reference_manual/cc/f9/93/b2/f0/82/42/57/CD00240193.pdf/files/CD00240193.pdf/jcr:content/translations/en.CD00240193.pdf.

[4] Stmicroelectronics. PM0056: STM32F10xxx/20xxx/21xxx/L1xxxx Cortex-M3 programming manual [EB/OL]. (2017-12-20)[2018-02-01]. https://www.st.com/content/ccc/resource/technical/document/programming_manual/5b/ca/8d/83/56/7f/40/08/CD00228163.pdf/files/CD00228163.pdf/jcr:content/translations/en.CD00228163.pdf.

[5] ARM. Cortex-M3 Devices Generic User Guider[EB/OL]. [2018-02-01]. http://infocenter.arm.com/help/topic/com.arm.doc.dui0552a/DUI0552A_cortex_m3_dgug.pdf.

清华科技大讲堂
前沿·科技·分享

算法竞赛入门到进阶

- 算法思路　一点就透，豁然开朗
- 模板代码　结构精巧，清晰易读
- 知识体系　由浅入深，逐步推进
- 赛事相关　参赛秘籍，高手经验

微课版

120分钟视频讲解

罗勇军 郭卫斌 ◎ 著

清华大学出版社

清华科技大讲堂

算法竞赛入门到进阶

罗勇军　郭卫斌　著

清华大学出版社
北　京

内 容 简 介

本书是算法竞赛的入门和进阶教材,包括算法思路、模板代码、知识体系、赛事相关等内容。本书把竞赛常用的知识点和竞赛题结合起来,讲解清晰、透彻,帮助初学者建立自信心,快速从实际问题入手,模仿经典代码解决问题,进入中级学习阶段。

全书分为12章,覆盖了目前算法竞赛中的主要内容,包括算法竞赛概述、算法复杂度、STL和基本数据结构、搜索技术、高级数据结构、基础算法思想、动态规划、数学、字符串、图论、计算几何。

本书适合用于高等院校开展的ICPC、CCPC等算法竞赛培训,中学NOI信息学竞赛培训,以及需要学习算法、提高计算思维的计算机工作者。

图书在版编目(CIP)数据

算法竞赛入门到进阶/罗勇军,郭卫斌著.—北京:清华大学出版社,2019
(清华科技大讲堂)
ISBN 978-7-302-52915-6

Ⅰ.①算… Ⅱ.①罗… ②郭… Ⅲ.①计算机算法 Ⅳ.①TP301.6

中国版本图书馆CIP数据核字(2019)第083538号

策划编辑: 魏江江
责任编辑: 王冰飞
封面设计: 刘 键
责任校对: 李建庄
责任印制: 杨 艳

出版发行: 清华大学出版社
网 址: http://www.tup.com.cn, http://www.wqbook.com
地 址: 北京清华大学学研大厦A座 **邮 编:** 100084
社 总 机: 010-62770175 **邮 购:** 010-62786544
投稿与读者服务: 010-62776969, c-service@tup.tsinghua.edu.cn
质量反馈: 010-62772015, zhiliang@tup.tsinghua.edu.cn
课件下载: http://www.tup.com.cn,010-62795954
印 刷 者: 北京富博印刷有限公司
刷 订 者: 北京市密云县京文制本装订厂
经 销: 全国新华书店
开 本: 185mm×260mm **印 张:** 22.5 **字 数:** 546千字
版 次: 2019年8月第1版 **印 次:** 2019年8月第1次印刷
印 数: 1~2500
定 价: 59.80元

产品编号:081639-01

推　荐　序

大学生算法竞赛，例如ICPC（国际大学生程序设计竞赛）、CCPC（中国大学生程序设计竞赛），是目前中国最具影响力的大学生计算机赛事。

这些算法竞赛之所以具有很大的影响力，是因为它们综合考察了参赛队员在编码能力、算法知识、逻辑思维、创新能力、团队合作等各方面的素质。很多从算法竞赛中走出的获奖学生已陆续成长为杰出的软件工程师。近年来，在国内、国际闻名的IT公司中，有算法竞赛背景的创业者也层出不穷。

我长期从事算法竞赛的培训工作，深刻认识到算法竞赛在我国IT教育中的关键作用。可以说，IT行业决定了一个国家的未来发展高度。目前，中国的IT行业已经在世界上颇具影响力。中国每年培养约100万信息类大学生，但仍然供不应求。特别是高级软件工程师，更是各大企业、创业公司争抢的对象。算法竞赛的目标正是培养杰出的软件工程师，并且中国20多年的算法竞赛历史已经证明，算法竞赛培训是非常有效的手段。

随着算法竞赛的发展，全国大部分高校陆续开展了算法竞赛的课内或课外课程，各大学的竞赛指导老师为之付出了极大的心血。

算法竞赛不同于其他的学科竞赛，它的长期性、艰苦性使它成为一个高难度的学习活动。参赛队员需要长期持之以恒地学习、训练，指导老师也需要在竞赛培训、教材编写、日常管理、组织参赛、主办比赛、维护OJ系统等方面做大量琐碎而艰苦的工作。竞赛教材的编写是其中一项重要内容，近些年，国内也陆续出版了十几种相关教材，这些都是算法竞赛学生日常学习的基本资料。

我的同行罗勇军老师是华东理工大学的竞赛教练，长期从事算法竞赛的指导工作。我和罗老师很早就认识，虽然不常见面，但网络交流很频繁，经常就竞赛学生的有效管理模式进行经验交流。

令我印象深刻的是，2008年我校（杭州电子科技大学）主办了ACM-ICPC亚洲区域赛，罗老师带领的参赛队一举获得金牌，并因此入围了次年于瑞典举办的第33届ACM-ICPC World Final，这让当时的我既羡慕，又深受鼓舞。一年后，我校也实现了Final的历史性突破，并在接下来的几年中先后5次入围全球总决赛。

罗老师作为一名十几年坚持在教学第一线的金牌教练，在总结长期的算法竞赛教学经验的基础上写下了《算法竞赛入门到进阶》一书。通读下来，这本书的语言描述贴近学生的阅读习惯，有很好的可读性，对基础算法的讲解详细、清晰、通俗易懂；仔细读之，读者能够深刻感受到作者为之付出的心血。

书中的例题大部分来自杭州电子科技大学在线提交系统（HDOJ），这让我倍感亲切。

同时我想，这对于国内的读者来说也一定是乐于看到的，毕竟绝大部分参加竞赛的学生都是HDOJ的用户。

总之，希望罗老师的这本《算法竞赛入门到进阶》能给广大的参赛学生带来实实在在的帮助。

刘春英
杭州电子科技大学
2019年6月

前　　言

算法竞赛，例如ACM-ICPC、CCPC等，在中国已经活跃多年，是最具影响力的大学生计算机竞赛。目前，已经出版的算法竞赛书也有30多部，有一些被队员们奉为“宝书”，有很好的口碑。本书作者是竞赛教练，因为工作的原因，详细阅读过这些书。这些书，或者讲解深刻让人佩服，或者娓娓道来令人愉悦，或者洋洋大观让人欲罢不能。读经典书，甘之如饴。

在多年的竞赛教练工作中，本书作者作为喜欢自我表现的社会人，也常常跃跃欲试，试图写出一本新的经典书。本书作者认为，竞赛队员在算法竞赛学习中的痛点需求如下。

算法思路：一点就透，豁然开朗。

模板代码：结构精巧，清晰易读。

知识体系：由浅入深，逐步推进。

赛事相关：参赛秘籍，高手经验。

上面立的几个flag虽然高不可攀，但确实是本书作者内心的旗帜。

本书是一本“竞赛书”，不是计算机算法教材，也不是编程语言书，因此对大多数知识点本身不会做过多的讲解，而是把重点放在讲解竞赛所常用的知识点上，以及如何把知识点和竞赛题结合起来。当然，由于编程竞赛涉及太多知识点，一本竞赛书不可能面面俱到，把所有内容都堆砌进来。市面上还有太多经典的算法教材和编程语言教材，这都是竞赛队员应该认真阅读的。

本书对知识点进行了精心的剖析。很多知识点看起来复杂难解，但如果结合清晰的代码、生动的文字、通俗的比喻、一目了然的图解、画龙点睛的注解，就能让人豁然开朗。这也是本书的目标。

代码能力体现了编程者的实力。学习别人的好代码是提高自己编码水平的捷径。本书把知识点讲解和竞赛题目紧密地结合在一起，同时给出实用的代码。这些代码有的是作者精心组织和编写的，有的是搜索大量资料后进行整理总结的结果。其中很多代码完全可以作为编程的模板，希望能对参赛学生起到参考的作用。特别是经典问题，往往有经典代码，凝结了很多人的劳动。本书作者并没有独创经典代码的能力，因此书中不可避免地引用和改写了一些公开的代码。对于一些能找到出处的经典代码，在书中都标注了出处。

本书主要面向初学者和中级进阶者。初学者面对海量繁杂的竞赛知识点往往会产生深深的无力感和挫折感，本书由浅入深地讲解知识点，逐步推进，帮助初学者建立自信心，从而快速地从能理解的实际问题入手，模仿经典代码解决问题，进入中级学习阶段。

竞赛是很专业的活动，经验非常重要。书中就一些日常训练和参赛的细节问题介绍了作者的体会。

学习算法竞赛有很大难度，需要精通编程语言、掌握很多算法，但是这并不意味着需要先学好算法和编程语言才能进行竞赛训练。事实上，建议初学者从零基础就开始学习算法

编程竞赛,与算法学习和语言学习同步进行。竞赛是操练的擂台,竞赛题目把知识点和具体问题结合起来,让学到的知识有了打击的“力点”。

以上是本书的特点,希望本书能给算法竞赛的初学者和进阶学习者以较大的帮助。如果是初学者,通过本书可以快速入门,例如了解竞赛的知识点、建立算法思维、动手写出高效率的代码。如果是中级进阶者,学习本书,可以更透彻地掌握复杂算法的思想、学习经典代码、完善知识体系,从而更自信地加入到竞争激烈的比赛活动中。

本书提供教学大纲、教学课件、程序源码,扫描封底的课件二维码可以下载;本书还提供120分钟的视频讲解,扫描书中的二维码可以在线观看。

在本书的编写过程中,华东理工大学竞赛队员提出了一些建议,感谢2015级队长姚远,以及王亦凡、王泽宸、翁天东、傅志凌等队员。

作　者

2019年5月

目　　录


源码下载

第 1 章　算法竞赛概述 ………………………………………… 1

1.1　培养杰出程序员的捷径 ………………………………………… 2
1.1.1　编写大量代码 ………………………………………… 3
1.1.2　丰富的算法知识 ………………………………………… 3
1.1.3　计算思维和逻辑思维 ………………………………………… 3
1.1.4　团队合作精神 ………………………………………… 3
1.2　算法竞赛与创新能力的培养 ………………………………………… 4
1.3　算法竞赛入门 ………………………………………… 5
1.3.1　竞赛语言和训练平台 ………………………………………… 5
1.3.2　判题和基本的输入与输出 ………………………………………… 5
1.3.3　测试 ………………………………………… 8
1.3.4　编码速度 ………………………………………… 9
1.3.5　模板 ………………………………………… 10
1.3.6　题目分类 ………………………………………… 11
1.3.7　代码规范 ………………………………………… 12
1.4　天赋与勤奋 ………………………………………… 13
1.5　学习建议 ………………………………………… 14
1.6　本书的特点 ………………………………………… 15

第 2 章　算法复杂度 ………………………………………… 17

2.1　计算的资源 ………………………………………… 17
2.2　算法的定义 ………………………………………… 21
2.3　算法的评估 ………………………………………… 22

第 3 章　STL 和基本数据结构 ………………………………………… 24

3.1　容器 ………………………………………… 24
3.1.1　vector ………………………………………… 25
3.1.2　栈和 stack ………………………………………… 27
3.1.3　队列和 queue ………………………………………… 28
3.1.4　优先队列和 priority_queue ………………………………………… 30
3.1.5　链表和 list ………………………………………… 30

3.1.6 set …… 31
3.1.7 map …… 33
3.2 sort() …… 34
3.3 next_permutation() …… 35

第4章 搜索技术 …… 37

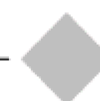
4.1 递归和排列 …… 38
4.2 子集生成和组合问题 …… 41
4.3 BFS …… 43
4.3.1 BFS和队列 …… 43
4.3.2 八数码问题和状态图搜索 …… 46
4.3.3 BFS与A*算法 …… 50
4.3.4 双向广搜 …… 52
4.4 DFS …… 53
4.4.1 DFS和递归 …… 53
4.4.2 回溯与剪枝 …… 54
4.4.3 迭代加深搜索 …… 57
4.4.4 IDA* …… 58
4.5 小结 …… 60

第5章 高级数据结构 …… 61

5.1 并查集 …… 62
5.2 二叉树 …… 66
5.2.1 二叉树的存储 …… 66
5.2.2 二叉树的遍历 …… 67
5.2.3 二叉搜索树 …… 70
5.2.4 Treap树 …… 72
5.2.5 Splay树 …… 78
5.3 线段树 …… 84
5.3.1 线段树的概念 …… 84
5.3.2 点修改 …… 85
5.3.3 离散化 …… 89
5.3.4 区间修改 …… 90
5.3.5 线段树习题 …… 93
5.4 树状数组 …… 93
5.5 小结 …… 97

第6章 基础算法思想 …… 98

6.1 贪心法 …… 98



6.1.1 基本概念 …… 98
6.1.2 常见问题 …… 100
6.1.3 Huffman 编码 …… 102
6.1.4 模拟退火 …… 105
6.1.5 习题 …… 107
6.2 分治法 …… 107
6.2.1 归并排序 …… 108
6.2.2 快速排序 …… 111
6.3 减治法 …… 113
6.4 小结 …… 114

第7章 动态规划 …… 115

7.1 基础 DP …… 116
7.1.1 硬币问题 …… 116
7.1.2 0/1 背包 …… 123
7.1.3 最长公共子序列 …… 127
7.1.4 最长递增子序列 …… 129
7.1.5 基础 DP 习题 …… 132
7.2 递推与记忆化搜索 …… 133
7.3 区间 DP …… 134
7.4 树形 DP …… 139
7.5 数位 DP …… 144
7.6 状态压缩 DP …… 148
7.7 小结 …… 153

第8章 数学 …… 154

8.1 高精度计算 …… 154
8.2 数论 …… 155
8.2.1 模运算 …… 156
8.2.2 快速幂 …… 156
8.2.3 GCD、LCM …… 159
8.2.4 扩展欧几里得算法与二元一次方程的整数解 …… 159
8.2.5 同余与逆元 …… 161
8.2.6 素数 …… 163
8.3 组合数学 …… 166
8.3.1 鸽巢原理 …… 166
8.3.2 杨辉三角和二项式系数 …… 167
8.3.3 容斥原理 …… 168
8.3.4 Fibonacci 数列 …… 168

8.3.5 母函数 …… 169
8.3.6 特殊计数 …… 174
8.4 概率和数学期望 …… 180
8.5 公平组合游戏 …… 183
8.5.1 巴什游戏与 P-position、N-position …… 184
8.5.2 尼姆游戏 …… 185
8.5.3 图游戏与 Sprague-Grundy 函数 …… 187
8.5.4 威佐夫游戏 …… 190
8.6 小结 …… 191

第 9 章 字符串 …… 192

9.1 字符串的基本操作 …… 192
9.2 字符串哈希 …… 194
9.3 字典树 …… 196
9.4 KMP …… 198
9.5 AC 自动机 …… 202
9.6 后缀树和后缀数组 …… 204
9.6.1 概念 …… 205
9.6.2 用倍增法求后缀数组 …… 206
9.6.3 用后缀数组解决经典问题 …… 212
9.7 小结 …… 213

第 10 章 图论 …… 214

10.1 图的基本概念 …… 214
10.2 图的存储 …… 215
10.3 图的遍历和连通性 …… 217
10.4 拓扑排序 …… 219
10.5 欧拉路 …… 223
10.6 无向图的连通性 …… 225
10.6.1 割点和割边 …… 225
10.6.2 双连通分量 …… 228
10.7 有向图的连通性 …… 230
10.7.1 Kosaraju 算法 …… 231
10.7.2 Tarjan 算法 …… 234
10.8 2-SAT 问题 …… 236
10.9 最短路 …… 239
10.9.1 Floyd-Warshall …… 240
10.9.2 Bellman-Ford …… 242
10.9.3 SPFA …… 246



10.9.4 Dijkstra …… 250
10.10 最小生成树 …… 253
10.10.1 prim 算法 …… 254
10.10.2 kruskal 算法 …… 255
10.11 最大流 …… 257
10.11.1 Ford-Fulkerson 方法 …… 258
10.11.2 Edmonds-Karp 算法 …… 260
10.11.3 Dinic 算法和 ISAP 算法 …… 262
10.12 最小割 …… 263
10.13 最小费用最大流 …… 264
10.14 二分图匹配 …… 268
10.15 小结 …… 271

第 11 章 计算几何 …… 272

11.1 二维几何基础 …… 272
11.1.1 点和向量 …… 273
11.1.2 点积和叉积 …… 274
11.1.3 点和线 …… 276
11.1.4 多边形 …… 280
11.1.5 凸包 …… 283
11.1.6 最近点对 …… 285
11.1.7 旋转卡壳 …… 287
11.1.8 半平面交 …… 288
11.2 圆 …… 293
11.2.1 基本计算 …… 293
11.2.2 最小圆覆盖 …… 297
11.3 三维几何 …… 300
11.3.1 三维点和向量 …… 300
11.3.2 三维点积 …… 301
11.3.3 三维叉积 …… 302
11.3.4 最小球覆盖 …… 304
11.3.5 三维凸包 …… 304
11.4 几何模板 …… 308
11.5 小结 …… 315

第 12 章 ICPC 区域赛真题 …… 316

12.1 ICPC 亚洲区域赛(中国大陆)情况 …… 316
12.2 ICPC 区域赛题目解析 …… 317
12.2.1 F 题 Friendship of Frog(hdu 5578) …… 318

12.2.2 K 题 Kingdom of Black and White(hdu 5583) …………………… 320
12.2.3 L 题 LCM Walk(hdu 5584) ………………………………………… 323
12.2.4 A 题 An Easy Physics Problem(hdu 5572)……………………… 325
12.2.5 B 题 Binary Tree(hdu 5573) ………………………………………… 326
12.2.6 D 题 Discover Water Tank(hdu 5575)…………………………… 328
12.2.7 E 题 Expection of String(hdu 5576) ……………………………… 333
12.2.8 G 题 Game of Arrays(hdu 5579) ………………………………… 336
12.2.9 I 题 Infinity Point Sets(hdu 5581) ……………………………… 339

参考文献 ……………………………………………………………………………… 344

第 1 章　算法竞赛概述


视频讲解

✍算法竞赛简介
✍创新能力的培养
✍训练平台
✍入门知识
✍模板的作用
✍题目分类
✍学习计划

算法竞赛是培养大学生程序设计能力、计算思维能力、创新能力和团队合作精神的重要方式，是培养杰出程序员的捷径，被国内高校普遍重视，吸引着越来越多的大学生参与其中。

本章介绍了与竞赛相关的入门知识，包括竞赛语言及训练平台、编码规范、题目分类、模板的作用等。在本章的最后部分讨论了天赋和努力与竞赛成绩的辩证关系，并详细地给出了学习建议。

算法竞赛(程序设计竞赛)是培养杰出程序员的捷径。

在当前高等教育强化创新能力培养、逐年增大学科竞赛投入的背景下，出现了一大批面向大学生的算法类竞赛。在国内众多竞赛中，最具影响力的面向大学生的程序设计竞赛有 ACM-ICPC 和 CCPC 等，面向中学生的程序设计竞赛有全国青少年信息学奥林匹克竞赛。

ACM-ICPC(International Collegiate Programming Contest，国际大学生程序设计竞赛)[①]是国际和国内最有影响力的高校计算机竞赛，是旨在展示大学生创新能力、团队精神和在压力下编写程序、分析和解决问题能力的年度竞赛。经过多年的发展，ACM-ICPC 已经成为全球最具影响力的大学生程序设计竞赛，它也成为我国高校创新教育评估的主要竞赛之一。

CCPC(China Collegiate Programming Contest，中国大学生程序设计竞赛)[②]是由中国大学生程序设计竞赛组委会主办的面向世界大学生的国际性年度赛事，旨在激发学生学习算法和程序设计的兴趣，提升算法设计、逻辑推理、数学建模、编程实现和英语阅读能力，激励学生运用计算机编程技术和技能解决实际问题，培养团队合作意识、挑战精神和创新潜力。中国大学生程序设计竞赛是 ACM-ICPC 在中国发展的必然产物。

---

① 网址为 icpc. baylor. edu。从 2018 年起，ACM 协会不再赞助 ICPC，因此众所周知的 ACM-ICPC 竞赛应该改为 ICPC。本书由于需要介绍竞赛的历史，这里仍然沿用 ACM-ICPC 这个名称。

② 网址为 ccpc. io。

NOI(National Olympiad in Informatics,全国青少年信息学奥林匹克)[①]是国内省级代表队最高水平的大赛。每年各省选拔产生5名选手,由中国计算机学会组织进行比赛。

"学科竞赛不仅是高校创新人才培养的重要手段,而且是用人单位选拔人才的重要依据"[②],算法竞赛完全证明了这一点。搜索所有著名IT公司招聘软件工程师的面试题目就会发现,没有经过算法竞赛训练的人很难通过这些面试。

算法竞赛是展示编程能力的大舞台。练武的人,如果有一个比武的擂台,可以极大地激发他们的活力,并让他们有机会证明自己不是纸上谈兵,而是真正的高手。ACM-ICPC、CCPC、NOI就是学习编程和展示程序设计能力的大擂台。在学习的过程中,从简单题目到难题,从一个专题到所有专题,一步一步渐进学习,让参与者能清晰地把握自己的进步。竞赛题目都是实打实的软件小项目,完成它们能给人以真实的成就感,并验证是否掌握了真正的编程本领。

1.1 培养杰出程序员的捷径

杰出的程序员有什么特质?这里可以列出一长串:掌握多种编程语言,编写过大量代码,算法知识丰富,数学应用能力强,做过很多项目,有团队精神和创新意识,善于根据行业需求调整自己的努力方向……

学习和参加算法竞赛是成为杰出程序员的捷径。ACM-ICPC的冠军被称为"世界上最聪明的人",竞赛的获奖者基本上都成长为杰出的软件工程师,并且有很多人是IT公司的创业者。例如,当前最热门的人工智能公司,很多创始人都是算法竞赛的佼佼者。

依图科技联合创始人林晨曦,2002年获ACM-ICPC总决赛金牌,2008年进入阿里云任技术总监,搭建中国首个拥有自主知识产权的分布式计算平台"飞天",2012年与同学朱珑一起联合创立依图科技。

第四范式CEO戴文渊,2005年获ACM-ICPC总决赛金牌,在"《科学中国人》2016年度人物"评选中,成为第一位代表人工智能行业获评"年度科技型企业家"荣誉的企业家。

旷视科技的创始人唐文斌,2008年获ACM-ICPC总决赛银牌,他的公司被李开复称为"国内最成功的人工智能公司"。旷视科技公司聚集了很多算法竞赛获奖的人才。唐文斌这样评价自己公司的员工:"在我的团队里,聚集了一批这样的天才人物。目前旷视科技团队成员有60个左右,其中30多个人至少曾获得过一项世界级编程比赛奖项,得过国际奥林匹克竞赛金牌的有7个……"[③]

在ACM-ICPC区域赛上获奖的学生,在大学生中比例极小。例如,2017年亚洲区域赛,参赛队员是从300个大学选拔出来的,7个赛区合计约1700队。其中,金牌10%,170队,约500人,以大四学生为主;银牌20%,340队,约1000人,以大三、大四学生为主;铜牌30%,500队,约1500人,以大三学生为主。这其中有一部分队伍重复参加多个赛区的比

---

① 网址为www.noi.cn。
② 中国高等教育学会《高校竞赛评估与管理体系研究》专家工作组。
③ 唐文斌:先打地基再建楼:http://m.iheima.com/article/150664(永久网址:perma.cc/QAE8-NKVX)。

赛，估算起来，每年毕业的金牌获奖者不到500人，银牌获奖者不到1000人。中国在校的计算机类专业大学生有100万左右，这还不算其他信息类专业毕业生。因此，ACM-ICPC竞赛获奖队员可以说是千里挑一、万里挑一的杰出人才了。

在大学阶段参加算法竞赛，可以使一个未来的杰出程序员获得下面几个小节所介绍的能力。这些能力虽然被视为"基础能力"，但却是大部分学计算机编程的学生所不能轻易获得的。

1.1.1　编写大量代码



视频讲解

比尔·盖茨曾说过："如果你想雇用一个工程师，看看他写的代码就够了。如果他没写过大量代码，就不要雇用他。"[①]通过编写大量代码，能做到算法精妙合理、逻辑清晰透彻、代码喷涌而出、格式赏心悦目、挑bug手到擒来，这是杰出程序员的基本功。ACM-ICPC竞赛队员想达到在区域赛中获奖的水平，需要写5～10万行的代码。

1.1.2　丰富的算法知识

算法是程序的核心，决定了程序的优劣。特别是在数据规模大的情况下，算法直接决定了程序的生死。例如，用计算机处理排序问题：假设有100万个数，用最简单的冒泡排序算法，计算量可能多达1万亿次（冒泡排序的计算复杂度是$O(n^2)$，100万×100万＝1万亿），在计算机上，计算时间长达几个小时，实际上根本不能用；如果改用快速排序算法，计算量只有2000万次（快速排序的计算复杂度是$O(n\log_2 n)$，100万×$\log_2$100万≈2000万），计算机在1秒内可以完成。二者的计算时间相差5万倍，算法的威力可见一斑。

算法竞赛涉及绝大部分常见的确定性算法，掌握这些知识，不仅能应用在软件开发中，也是进一步探索未知算法的基础。例如现在非常火爆的、代表了人类未来技术的人工智能研究，涉及许多精深的算法理论，没经过基础算法训练的人根本无法参与。

1.1.3　计算思维和逻辑思维

一些竞赛队员经常说：我们要尽量掌握所有算法知识。但是，程序设计不仅要有算法思想，还需要能正确地写出程序，这不是仅仅有算法知识就能完成的。一道难题，往往需要综合多种能力，例如数据结构、算法知识、数学方法、流程和逻辑等，这就是计算思维和逻辑思维能力的体现。通常，能解出这样的题目是高级程序员的特征。

在ACM-ICPC亚洲区域赛和CCPC赛事上获得金牌、银牌的队员，能够凭借奖牌证实自己有这样的能力。这也是算法竞赛被看重的主要原因。

1.1.4　团队合作精神

在软件行业，团队合作非常重要，这一点不需要更多说明。ACM-ICPC、CCPC竞赛把对团队合作的要求放在了重要位置。竞赛的赛制是3人一队，一台计算机，十几道竞赛题，

---

① Gates："If you want to hire an engineer, look at the guy's code. That's all. If he hasn't written a lot of code, don't hire him."；https://www.wired.com/2010/04/ff_hackers/。

限定 5 个小时。参加过现场比赛的队伍都能立刻体会到：一个队伍的 3 个人，在同等水平下，如果配合默契，则可以多做一两道题，把获奖等级提高一个档次。在竞赛过程中，有人负责精读英语题，有人负责构造测试数据，有人负责编写代码，大家互相讨论思路，队长判断现场形势，确定做题顺序。每支队伍的 3 个人只有在日常训练中长期磨合，才能互相了解，做到合理分工、优势互补，从而发挥出最优的团队力量。

有人认为：毕业后参加工作，其实用不着算法竞赛这么多的复杂逻辑和算法知识，即使用到了，在工程中一般有现成的模块，拿来用就行了，只要了解这个模块用到的算法的作用和复杂度即可。这种认识是肤浅的，对于立志成为高级程序员的学生而言，进行大量的计算思维训练和经典算法训练是必需的，理由如下：

(1) 算法是对学习和理解能力的一块试金石，难的都能掌握，容易的当然不在话下。在算法竞赛上获奖的人证明了自己有解决复杂编程问题的能力。

(2) 即使有现成的模块，但是对于特定的需求，往往需要进行修改才能真正使用，没有真正理解的人无法修改。

(3) 实际的程序往往有复杂的逻辑关系，但又不属于经典的算法，没有现成模块，需要自己思考才能写出代码，这个能力是通过训练得到的。

1.2 算法竞赛与创新能力的培养

算法竞赛培养这样的能力：对复杂问题，用高效的算法或逻辑进行建模并编码实现。

目前中国的 IT 业极其繁荣，已经和美国并列为世界超级两强，把其他国家和地区远远抛到后面，并且中国有加速超过美国的趋势。繁荣意味着竞争激烈，参加编程竞赛的获奖队员能够在成千上万的 IT 工程师中脱颖而出，有很多创业并成功。本书作者认为，他们需要具备以下品质：

(1) 激情和勇气。例如，一旦开始，就不肯退缩的激情；渴望成为大人物，有改变行业的野心；敢于平等地和老师、IT 行业人士进行交流的勇气。本书作者所在的华东理工大学有一些竞赛队员毕业后创业成功，他们在大一的时候就已经表现出了这些特点。例如创办杭州美登科技有限公司(淘宝的金牌“淘拍档”)的 2008 届毕业生邹宇、创办上海萌果信息科技有限公司(中国手游企业的明星)的 2009 届毕业生尹庆，在大一的时候就表现出了不服输、敢于承担的特点，他们先后担任了竞赛队长。

(2) 开阔的思路。能抓住一切机会了解更多的信息。例如创办云片网络的华东理工大学的 2007 届毕业生刘大林，他在读大三的时候发现学校主页没有校内搜索功能，于是主动做了一个搜索，并推销给了学校。再如华东理工大学的 2012 届毕业生诸咏天，在大学期间访问了很多创业公司，开阔了思路，在大学期间就积极创业，荣获“2011 年上海市大学生年度人物”的称号，毕业后创业并取得成功。

(3) 超群的技术能力。这一点非常重要。由于现在 IT 行业早已进入成熟阶段，从业者人数太多，竞争激烈，所以只有具备超群技术能力的人才能快速开发出难以模仿的软件，增加获得成功的概率。

(4) 自信。自信不是盲目的自大，自信的获得建立在征服困难的经验上。例如，参加过

ACM-ICPC 和 CCPC 竞赛的学生，由于经历了长期的非常困难的学习，在编程能力上远远超越了大部分学生，从而建立了超强的自信。

(5) 团队建设。竞赛的队员，在长期的共同学习和参赛的过程中团结合作、共同进步，结下了深厚的友谊，是未来创业的好伙伴。

1.3 算法竞赛入门

1.3.1 竞赛语言和训练平台

ICPC 允许的算法竞赛用的编程语言有 C、C++、Java、Python、Kotlin[①] 几种。其中 C++因运行效率高、具有丰富的 STL 函数库，最受竞赛队员欢迎。Java 和 Python 也比较常用，它们在处理大数据时极为简便，这几年 Python 上升的势头很快。这几种编程语言，在就业市场上都有大量的岗位需求，极容易就业。熟练掌握一种编程语言是基本的，掌握几种语言是必要的。


视频讲解

竞赛队员主要的学习方法就是"刷题"，在 Online Judge(OJ，在线判题网站)上大量做编程题。OJ 上有丰富的编程题目，能对编程者提交的程序进行自动判题，返回"正确"或"错误"提示。国内、国外有很多 OJ，国内的例如 acm. hdu. edu. cn、poj. org，国外的例如 uva、ural、usaco 等[②]。OJ 的核心价值主要有两个，即题目和判题用的测试数据。测试数据的重要性不亚于题目本身，甚至更重要。

很多队员在高中接触了 NOI 信息学竞赛，他们常常在 CCF 的 OJ[③]，以及"洛谷"和"大视野[④]"几个网站做题，以中文题为主。其中，"洛谷试炼场[⑤]"的题目分类比较全，是很好的基础学习平台。

在这些 OJ 之外还有一些代理 Judge。这些代理以 http 的方式调用了宿主 OJ 提供的判题服务，连接了 30 多个著名的 OJ，相当于一个综合平台。代理 Judge 的优点如下：

(1) 方便做国外题目，因为在国内直接连接外国的 uva 等 OJ 往往极慢，而通过代理很快。

(2) 如果某个 OJ 网站直接连不上，在代理上也常常能做这些 OJ 的题目。

(3) 虚拟比赛功能，即把来自不同宿主 OJ 的题目混编为一场训练赛，特别方便日常训练[⑥]。

搭建 OJ 系统在技术上是不难的。事实上，几乎所有常年开展算法竞赛的学校都建立了自己的 OJ，用于训练和比赛。

1.3.2 判题和基本的输入与输出

在 OJ 提交程序后，OJ 如何判断程序是正确的还是错误的？

---

① https://icpc. baylor. edu/worldfinals/programming-environment(短网址：t. cn/Rx4xYSH)

② 参考 cn. vjudge. net 列出的 OJ 网站。

③ 全国青少年信息学奥林匹克竞赛网站：www. noi. cn；做题网站：oj. noi. cn。

④ 大视野 OJ：www. lydsy. com，网上简称 bzoj。

⑤ 洛谷试炼场：https://www. luogu. org/training/mainpage，有不错的分类。

⑥ 专题学习：kuangbin 带你飞，vjudge. net/article/187。

OJ 由计算机自动判题,但计算机并没有看懂代码的智能;即使是人工判题,人也很难在短时间内看懂程序。因此,OJ 判题是一种黑盒测试,它并不关心程序的内容,而是用测试数据来验证。

OJ 的后台存储了每个题目的测试数据,有输入数据和对应的输出数据,并且有很多组输入和输出数据。OJ 运行用户提交的程序后读取输入数据,程序产生输出,然后与后台的标准输出进行对比,可以得到以下结果之一①:

(1) 没有超时,并且完全一致,判定 Accepted (AC)。

(2) 超时,判定 Time Limit Exceeded (TLE)。在一般情况下,返回 TLE 说明方法错误,整个程序可能需要推倒重来。

(3) 结果是对的,但是格式有错误,例如多了空格,返回 Presentation Error (PE)。

(4) 结果错误,或者有其他问题,返回 WA、RE、MLE 等信息。

由于 OJ 不看程序内容,只关心程序的输入和输出,所以在程序中不写详细过程,而是用 printf()或 cout 直接打印结果,这也是允许的,这种方法叫"打表"。另外,程序可能需要预处理数据,这个做法也称为"打表"。

一个题目的测试数据可能有成千上万组。好的测试数据应该尽量覆盖所有可能的情况,而不好的测试数据会让题目失去价值,这就是为什么测试数据和题目本身一样重要的原因。

1. 输入与输出函数

C++语言中的标准输入语句为 cin,输出语句为 cout②。

C 语言中的输入与输出函数如下。

- putchar():把一个字符常量输出到显示器屏幕上;
- getchar():从键盘上输入一个字符常量;
- printf():把数据按格式控制输出到显示器屏幕上;
- scanf():从键盘上输入各类数据;
- puts():把一个字符串常量输出到显示器屏幕上;
- gets():从键盘上输入一个字符串常量;
- sscanf():从一个字符串中提取各类数据。

在竞赛中,默认使用标准输入 stdin 和标准输出 stdout,所以在提交程序时并不用管 OJ 是怎么进行数据测试的。如果用到文件的输入与输出,会特别说明使用方法。

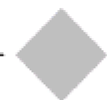
2. 输入结束方式

(1) 默认结束。在 OJ 上一般有多组测试数据,如果没有明确指出输入在什么时候结束,则程序以"文件结束"(EOF)为结束标志。例如:

```
int main(){
    int a,b;                                    //输入 a、b
```

① http://acm.hdu.edu.cn/faq.php。

② 在 *Competitive Programmer's Handbook*(作者 Antti Laaksonen,2017 年 10 月 11 日)的第 1 章中介绍了编程语言的一些注意事项。



```
    while(scanf("%d%d",&a,&b) != EOF){    //等价于 while(~scanf("%d%d", &a, &b)){
      …;
    }
    return 0;
}
```

一些队员喜欢把输入语句写成：

```
while(~scanf("%d%d", &a, &b))
```

这也是对的。因为如果没有输入，scanf()返回 EOF，系统定义 EOF = −1，取非就是 0。

在竞赛时，一般不建议用判断 EOF 的方法。本书的程序采用 while (~scanf("%d%d", &a, &b))形式。

(2) 在输入数据中指定了数据个数。一般在输入数据的第 1 行定义数据量大小，例如第 1 行是 100，则表示有 100 组数据。这里以 hdu 1090 题为例，程序如下：

hdu 1090 题程序

```
int main(){
    int n, a, b;
    scanf("%d", &n);                    //n: 有多少组数据
    while(n--){
        scanf("%d %d", &a, &b);
        printf("%d\n", a + b);
    }
    return 0;
}
```

(3) 以特定元素作为结束符。例如以 0 作为结束符，当输入读到 0 时就退出，可以这样写：

```
while(~scanf("%d",&n) && n)
```

3. 输入与输出的效率

在 C++语言中，输入和输出常用的语句是 cin、cout，优点是很方便。但是用户需要注意，与 scanf()、printf()相比，cin、cout 的效率很低，速度很慢。如果题目中有大量的测试数据，由于 cin、cout 输入和输出慢，可能导致 TLE，在这种情况下应使用 scanf()、printf()。

例如 hdu 3233 题。在本例中输入 $1 \leqslant T \leqslant 20\,000$，可能有 20 000 行数据，因此输入的效率很关键。此题用 scanf()、printf()可以 AC，OJ 返回的执行时间是 140ms；用 cin、cout，结果 TLE，执行时间超过 1000ms。

hdu 3233 程序：用 scanf()、printf()，AC，执行时间是 140ms

```
#include<bits/stdc++.h>
int main(){
    int T, n, cnt = 1, B;
    while(scanf("%d%d%d", &T, &n, &B)){
```

```
        if(T == 0 || n == 0 || B == 0) break;
        double s, p, sum = 0;
        while(T -- ) {                          //1≤T≤20000
            scanf("%lf%lf", &s, &p);            //高效率输入
            sum += s * (100 - p) * 0.01;
        }
        printf("Case %d: %.2f\n\n", cnt++, sum/(B * 1.0));
    }
    return 0;
}
```

hdu 3233 程序：用 cin 和 cout，结果 TLE，执行时间超过 1000ms

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    int T, n, cnt = 1, B;
    while(cin >> T >> n >> B){
        if(T == 0 || n == 0 || B == 0) break;
        double s, p, sum = 0;
        while(T -- ) {                          //1≤T≤20000
            cin >> s >> p;                      //输入很慢
            sum += s * (100 - p) * 0.01;
        }
        cout << "Case " << cnt++ << ": " << fixed << setprecision(2)
            << sum/(B * 1.0) << endl << endl;
    }
    return 0;
}
```

1.3.3 测试

1. 构造测试数据

在程序编好之后应该自己先测试通过，再提交到系统，而题目给的样例数据一般都太少，不足以检验程序的正确性，队员需要自己构造测试数据。

在一个队伍中，一般安排一个队员专门负责构造测试数据。对于需要高级算法的题目，可以让这名队员先用暴力法编程，然后随机生成输入数据，运行暴力程序，生成输出数据。输入数据除了随机生成以外，有时候还需要手工生成一些，主要是边界数据、特别小的数据、特别大的数据等，这些也是最容易出错的。

为方便操作，可以把构造出的输入数据放在文件 test.in 里，将程序的结果输出到文件 test.out 里。当然，有时候不需要输出文件 test.out，直接在屏幕上看输出结果就可以了。

那么如何方便地使用它们？有以下两种方法：

(1) 在程序中加入测试代码。

```
#define mytest
#ifdef mytest
```

```
    freopen("test.in", "r", stdin);
    freopen("test.out", "w", stdout);
#endif
```

在提交时，去掉 #define mytest 即可。

(2) 在行命令中重定向。

这是更简单的方法，不用在程序中加任何代码。例如，生成的可执行程序是 abc，在 Windows 或 Linux 的行命令中这样输入和输出到文件：

```
abc < test.in > test.out
```

2. 对比测试数据



视频讲解

对于复杂的题目，可能需要写两个程序：一个是提交到 OJ 的“好”程序；另一个是暴力法程序，目的是用它生成测试数据。这种方法称为“对拍”。

在测试的时候，可以用行命令比较两个程序的输出是否一致。例如在 Windows 系统下，生成的可执行文件分别是 abc.exe、abc_1.exe，用文件比较命令 fc 比较它们的输出是否一致。

```
abc.exe < test.in > test1.out
abc_1.exe < test.in > test2.out
fc test1.out test2.out /n
```

在 Linux 系统中，文件比较命令是 diff。

1.3.4　编码速度

竞赛时间很紧张，编码应该简洁。跟软件工程的代码相比，竞赛题的代码都不长，从几行到 200 多行。快速编程得到正确结果即可，无须担心程序是否符合工程项目的要求，也不要求写得多么“漂亮”，这是竞赛中编码的特点。

编程速度决定了参赛获奖的级别。在一般情况下，同样的出题数量会跨越相邻的获奖等级，例如同样做 5 道题，排前面的获银牌，排后面的获铜牌。但是，如果跨越的等级过大，则是不正常的。近些年来，由于区域赛的出题质量参差不齐，“速度”这个因素的影响越来越大。一套理想的题目应该有很好的区分度，例如金牌 8 题以上，银牌 6 题以上，铜牌 4 题以上。但是近年来常常遇到这样的赛区：终榜时，做题数量一样，只是因为出题快慢不同，名次就从银牌到铜牌再到铁牌（铁牌是 honorable mention，即鼓励奖的玩笑说法），分成了 3 个等级。应该说，这样大的跨越度不能区分参赛队伍的水平。

那么如何提高编码速度？

(1) 读题要快。题目都是英文的，新队员往往不习惯，需要长期训练才能适应。虽然有些小窍门，例如先读样例，再读题面内容，但最根本的还是靠大量的英语阅读练习，学会在脑海中直接用英语进行思考，才能提高读题速度。一套题需要 3 个队员分工快速读完，每个人读完后必须和队员一起讨论，确定完全理解题意。在竞赛现场紧张的气氛下，一个队员不要太相信自己，而忽视了队友的帮助。

(2) 熟练掌握编辑器或 IDE。根据规定，现场赛提供的编辑器有 vim、gedit 等，IDE 有

Eclipse、Code::Blocks 等[1]。Eclipse 和 Code::Blocks 受到新手和很多老队员的欢迎，不过一些老队员说，熟练使用编辑器 vim 写代码速度更快。在竞赛中，能赢得几分钟的领先时间有时是很关键的。据说，参加世界总决赛的队员使用 vim 的比例很高。

(3) 不要“霸占”计算机。由于竞赛时是三人一机，只能有一人使用键盘输入代码，另外两人在旁边手写代码，等计算机空闲了再使用。在平时训练时应该养成事先在纸上写好代码的习惯，不能边敲键盘边思考，否则会浪费机时。

(4) 减少调试。因为机时非常宝贵，所以除必要的代码输入和测试外尽量少使用计算机。在写好程序后，争取能一次通过测试样例。

为了减少调试，尽量使用不容易出错的方法，例如少用指针、使用静态数组、把逻辑功能模块化等。

另外，不要使用动态调试方法，不要用单步跟踪、断点等调试工具。如果需要查看中间数据，可用 cout()或 printf()打印出调试信息。

如果程序有问题，不要在计算机上检查，应该打印出来坐在旁边看，把机时让给队友。

(5) 互相检查。把代码讲给队友听是查错的好办法，即使队友不能理解你的思路，你在讲解的过程中也往往能突然发现自己的问题。

(6) 使用 STL。如果题目涉及比较复杂的数据处理，或者像 sort()这样需要灵活排序的函数，用 STL 可以大大减少编码量，并减少错误的发生。例如第 10 章的最短路径算法 Dijkstra，需要对结点进行松弛处理，自己编程实现会很烦琐，而如果直接使用 STL 的优先队列，编码能极大简化。

(7) 一些编码小技巧。例如把长字符重新定义成短字符，可以节省一点时间：

```
typedef  long  long  ll;
```

那么

```
long long a = 1234567890;
```

变成了简洁的：

```
ll a = 1234567890;
```

1.3.5 模板

使用模板对提高编码速度很有帮助。

刚参加算法竞赛学习的队员都听说有一种叫“模板”的神器。模板是参赛选手认为有用的代码片段，将其打印出来，允许带进竞赛现场作为“小抄”。在网上能找到很多老队员的模板，它们是学习编码的好的参考。

听起来“模板”是非常有用的：竞赛涉及几百种数据结构和算法知识，如果把它们的经典代码都总结出来，在做题的时候直接拿来用不就行了吗？这不就是软件工程的“模块化”吗？

---

[1] https://icpc.baylor.edu/worldfinals/programming-environment，规定了编程环境(短网址：t.cn/Rx4xYSH)。

现实也证明了这一点：有些赛区确实会出一些“模板题”，模板上的程序模块真的能直接应用在竞赛题中。

模板非常有用，其重要性主要在于帮助参赛选手理解经典算法，而不一定能用在赛场上。

使用模板需要考虑以下问题：

(1) 模板题并不常见。一个负责任的现场赛会避免出模板题，所有的题目都不能直接抄模板。

(2) 抄模板的能力。即使有模板，就一定会抄吗？模板的代码需要自己真正理解，并多次使用过，这样才能在做题的时候快速应用到编码中。不同的编程题目，即使用到相同的算法或数据结构，也往往不能用同样的代码，而需要做很多修改，因为不同环境下的变量和数据规模是不同的。因此，对模板的学习和使用需要花时间融会贯通，不能急躁。期望靠模板速成、急于拿去参赛获奖是没用的。

(3) 综合模板的能力。即使能用到模板，但是题目往往需要综合几个算法、逻辑、数据结构等，如何把模板融入整体代码中是很考验参赛选手能力的。如果只会用模板而没有理解模板，就像把尺寸不匹配的齿轮摁在一起，根本转不起来。

(4) 最重要的一点是**建模能力**。一个好题目符合这样的特征：题目很清晰，问题很清楚，然而很难想出它是什么算法、能用什么模板。但是，如果有人直接告诉你它其实是什么算法，可以用什么模板，你会恍然大悟，很快就能做出来。这个能力就是建模能力。真正的学习是掌握算法后面的思想，而不是只会背算法。通常有这样的比喻：若只会使用算法的模板而没有深入掌握算法的思想，则这个模板相当于残疾人的假肢，看起来像是自己的，走起来就知道不是自己的。只有把算法的思想深深根植于脑海，才会使其成为身体的一部分，达到心手一致的境界。从这个角度出发也能理解“质＞量”，在学习时不要追求学到的算法的“数量”，而是要掌握其“思想”。很多算法的思想其实是相通的。

本书所讲解的程序代码都经过了精心准备，是经典代码，大部分可以当模板学习。

每个队员都需要总结自己的模板。在比赛时带到赛场上，也许真的会遇到模板题呢！

提高编程速度，最根本的还是要通过大量练习，提高编码的熟练程度，“无他，但手熟尔！”

1.3.6 题目分类

算法竞赛涉及很多方面的知识，可以粗略地进行以下分类[①]：

- Ad Hoc，杂题；
- Complete Search (Iterative/Recursive)，穷举搜索(迭代/回溯)；
- Divide and Conquer，分治法；
- Greedy (usually the original ones)，贪心法；
- Dynamic Programming (usually the original ones)，动态规划；
- Graph，图论；
- Mathematics，数学；

① 在《Competitive Programming 3》(作者 Steven Halim、Felix Halim)的“1.2.2 Quickly Identify Problem Types”中。另外，在“1.4 The Ad Hoc Problems”中列出了大量杂题，读者可以做一做。

- String Processing，字符串处理；
- Computational Geometry，计算几何；
- Some Harder/Rare Problems，罕见问题。

除杂题外，其他分类都与数据结构和算法有关。

杂题也很常见，每次现场赛都会有 1 道或 2 道题。虽然程序设计竞赛的重点在算法方面，但是竞赛时有些题只考查逻辑能力和编码能力，并不涉及数据结构或算法，只要学过基本 C++语法就能做。这样的题目可能很难。作为练习，读者可以尝试下面的题目，它们都是大型模拟题，以烦琐、坑人著称，代码超过 200 行，需要很大的耐心和细心：

- bzoj 1972[1]“猪国杀”；
- bzoj 1033“杀蚂蚁”；
- bzoj 2548“灭鼠行动”。

本书内容包括杂题以外的所有分类，在每个分类中均会讲解常用的知识点。

1.3.7 代码规范

虽然说每个程序员都可以有自己的编码风格，但是还需要遵循大家公认的一些规范，以便于和队友互相交流。

下面列出了一些常见的注意事项。

(1) header。使用万能头文件“#include <bits/stdc++.h>”，OJ 网站一般都支持，一个例外是 poj，它不支持。

另外，不要用 C 风格的 header，例如 #include <stdio.h>。

(2) 输入判断结尾不要用 EOF，而用 '～'，例如：

```
~scanf("%d", &n)
```

在 1.3.2 节中已经详细介绍了这个问题。

(3) 换行。用 K&R 风格，即左大括号不换行，右大括号单列一行。

(4) 变量定义。变量定义在这个变量被调用的最近的地方，例如：

```
for (int i = 0; i < 10; i++) {              //i 只在这个循环体内使用
    int s = i * i;
}
```

(5) 最好不要用宏。不管是宏定义还是宏函数，都容易出问题。


视频讲解

不要用 #define 定义常量，而用 const 定义常量，例如：

```
const int MAX = 1000005;
```

把宏函数写成普通函数。

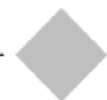
(6) 参考资料。

Google C++规范：https://google.github.io/styleguide/cppguide.html。

Linux C 规范：https://www.kernel.org/doc/html/v4.10/process/coding-style.html。

---

① https://www.lydsy.com/JudgeOnline/problem.php?id=1972(短网址：t.cn/RgCa1Si)。



1.4　天赋与勤奋

世上是否存在编程天才？答案是肯定的。普通智力能否达到很高的编程水平？答案也是肯定的。人们常说的"天赋决定上限，努力决定下限"，在编程竞赛这种高智力活动上有一定的道理。

天赋，在成功的因素中占有很大的比重。如果要达到顶级水平，天赋就更重要了。例如在体育运动项目中，能达到获得奥运奖牌水平的运动员，他的天赋几乎有决定性的影响。在脑力活动中，类似编程这样繁杂而高深的思维活动，智力的因素非常大。

如果读者有兴趣，可以用五子棋或魔方检验自己在记忆力、逻辑推理、空间想象力、专注度、敏捷性等方面的智力天赋。这两种游戏的特点是规则简单、上手快、变化比较复杂。所有人都能玩，但是想玩好，大部分人需要一段比较长的学习时间。一个初学者，如果只需要几天的学习就能达到很高的水平，那么差不多可以说他拥有编程天赋了[①②]。

这些有编程天赋的少数人，如果能专注练习，他们的学习效率要比普通人高出几倍，更容易成功。如果他们在刚上大学的时候从零基础开始学习编程，那么他们在大三甚至大二就能获得银牌，在大四或者大三就能获得金牌，可称为天之骄子！

智力普通的学生，通过勤奋的学习，挖掘出自己的智商潜力、锻炼自己的专业技能，也能达到很高的水平。特别是对于编程这种需要掌握海量知识、拥有长期编码经验的高智力活动来说，勤奋相对天赋的比重在职业生涯中会越来越大。根据经验，参加 ICPC 竞赛的学生，即使是零起点，如果能在大一入学后坚持每天 2～4 个小时的编程学习，那么他完全可以在大三的第一学期参加区域赛并获得铜牌，甚至银牌、金牌。

学习编程需要做好艰苦学习的心理准备。编程是一个长期、艰苦的过程，有乐趣，更有挫折。

在标题为"Why Learning to Code is So Damn Hard"的网页中[③]对学习编程的不同阶段给出了一个有趣的图，如图 1.1 所示。

该图把编程分成 4 个阶段，横坐标是编码能力，纵坐标是信心。

第 1 阶段(hand-holding honeymoon)：手把手关怀的蜜月期，能力和信心同步增长。初学者充满了乐趣，很有成就感，能找到丰富的学习资料。

第 2 阶段(cliff of confusion)：充满迷惑的下滑期。虽然编程者的实际能力在上升，但却逐渐丧失了信心。这是因为遇到了难以解决的问题、需要调试大量 bug、遇到挫败。不过这个时候仍能够找到答案，知识面也在变广。

第 3 阶段(desert of despair)：绝望的迷茫期，信心的沙漠。编程者遇到更加困难的问题，需要的知识剧增，但是资源匮乏，在网上也找不到答案，或者不知道该怎么提问，感觉就

---

① 一节课领悟五子棋：www.zhihu.com/question/265407029/answer/299115371(永久网址：perma.cc/uz27-BXKT)。

② 天才女程序员：www.zhihu.com/question/29784784(永久网址：perma.cc/XS4R-45ZS)。

③ www.vikingcodeschool.com/posts/why-learning-to-code-is-so-damn-hard(永久网址：perma.cc/BK4R-WS7F)这条曲线和 Dunning-kruger effect 的曲线很相似，与"认知偏差"有关。

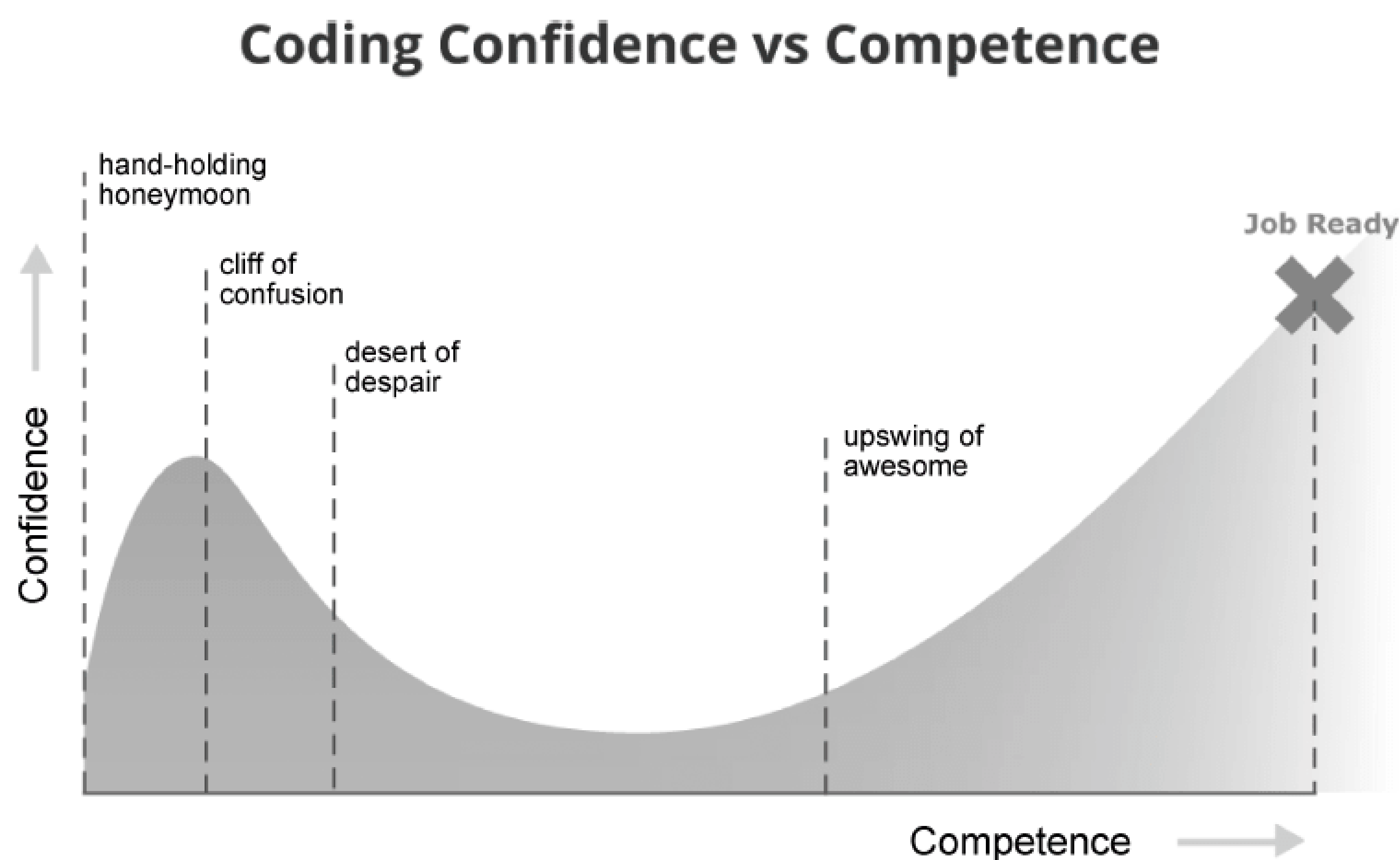


图 1.1　编码能力与信心的关系

像在沙漠一样。

第 4 阶段(upswing of awesome):煎熬的上升期。编程者心潮澎湃,浑身充满力量,绝望的沙漠已经过去。

简单地说,ICPC 区域赛铜牌水平的选手可能还未到达第 4 阶段;银牌以上水平的选手,可以确定到达了第 4 阶段,从而跨过了自由编程的门槛。

本书的内容,在这个图中估计只涉及整个阶段的前 30%,即前两个部分,也就是 hand-holding honeymoon 和 cliff of confusion,相当于入门和进阶。能否进入后两个阶段,取决于读者自己的努力。

1.5　学习建议

很多大学生在中学阶段就参加过 NOI 信息学竞赛,或者学习过编程,那么他们已经有了基础,进入大学后又投入了更多时间专心地进行编程训练,有这么好的起点当然是很有优势的。

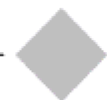
如果是完全的零基础,也不用担心自己落后。因为相比已经有了基础的同学只是晚学了几个月而已,只要多花一些时间,很快就能赶上。对于算法竞赛这样需要两三年的长周期学习来说,坚持才是最重要的。

由于算法竞赛的艰难和长期性,不管有没有基础,都应该从大一上学期开始学习。

(1) 大一上学期,熟悉 C、C++、Java 语言。一些专业在大一上学期开设编程语言课;大部分专业是在大一下学期,这些学生需要自学编程语言。

(2) 大一上学期,做一些简单的中文题,例如 acm.hdu.edu.cn 的 2000～2099 题[①]、洛谷试炼场。任务是进一步熟悉编程语言、学习如何在 OJ 上做题、掌握输入与输出的用法、

① http://acm.hdu.edu.cn/listproblem.php?vol=11,大部分比较简单,也有难题,可以跳过。



积累代码量。基本上每个题目在网上都能搜到题解和代码。初学者可以多看看别人的代码，尽快提高自己的编码能力。另外，最好几个人一起编程，并互相改错。看懂别人的代码，找出别人代码的错误，也是很好的训练，重要性**不亚于**独立做题。

（3）大一下学期，做一些入门题，例如搜索、数学、贪心、简单动态规划等，尽可能多地参加各校举办的新生网络赛。

（4）大一暑假，参加集训，学习数据结构、深入掌握 STL、进行各种专题入门，并熟悉队友。

（5）大二，深入各类专题学习，并制定一年的计划，牢固掌握各种算法知识点。如有可能，在大二上学期参加区域赛。

（6）大二暑假，组队参加网络赛和模拟赛。

（7）大三上学期，参加区域赛并获奖。

（8）大三和大四，开始难题、综合题的学习，使自己获得彻底的飞越，成为“编码大师”。通常，能获得金牌的队伍至少能做出 1 个以上的难题。难题有 3 个特征，即综合性强、思维复杂、代码冗长。这些难题是绝大部分学编程的学生难以翻越的大山，能征服大山的竞赛队员可以称为“杰出”了。

1.6　本书的特点

参加竞赛训练的队员需要阅读各种各样的算法书、编程书。本书作者不奢望写出一本面面俱到、老少咸宜的书，而且这样的书也许并不存在。本书面向的读者是初学者和中级进阶者，特点如下：

（1）算法知识点的讲解清晰、易懂。在众多确定的算法中，少数算法比较简单，多数比较难。通俗易懂地讲解一个复杂的算法是不容易的。对于初学者，经常发生的场景是在学习某个知识点的时候读了很多书，查阅了很多资料，却仍然头脑昏昏，不得要领，在痛苦地花了很多时间思索之后才恍然大悟。本书的编写目标是让读者“一点就透，豁然开朗”，因此，书中大量使用了比喻、图解、步骤、注解等方法，尽量降低初学者的学习难度。

（2）例题简单、直接。为了讲清楚算法，每个算法都需要配合竞赛题和代码，了解应用模型，并把理论和编码结合起来。本书选择的例题大多是简单、直接的“裸题”，很少有综合题、转弯题。也就是说，本书的目的是“奠基”，以及构建算法知识门类的“框架”，上面的高楼需要读者自己去建设。对于难题、综合题，已经有很多题解被编成书出版，读者也可以到网上搜索题解，网上的资源更加丰富。

（3）代码清晰、易读。准确、清晰的代码能让读者对算法知识的理解更加透彻。本书的每段代码都以成为“模板”为目标。这些代码是在借鉴大量代码的基础上提炼出来的，是作者精心总结的结晶。

（4）尽量覆盖竞赛的知识体系。虽然本书只能讲解部分常用的知识点，但是每一章都对相关知识点做了介绍，希望初学者在阅读本书的同时扩展本书未能讲解的知识。

总之，**本书不是一本题解**，而是一本**帮助读者建立计算思维**的书，旨在帮助读者打造坚实的基础，获得继续深入的信心。

最后，用下面的讨论作为本章的结语。

算法竞赛涉及的知识非常多，有些算法在竞赛中常用，有些不常用。如果初学者过于功利，就会纠结于哪些算法该学，哪些不该学。

作者的看法是，学习算法并不只是为了参加竞赛。每个经典算法都经过了无数人的精心研究，是极为精巧、合理的思维运动，是计算机科学这片天空的星星。学习它们，本身就是很好的思维练习，比做多少竞赛“热题”都强得多！

第2章　算法复杂度

✍计算的资源
✍算法的定义
✍算法的评估

编程初学者肯定思考过这样一个问题：拿到一个计算问题，自己尝试编程解决，衡量这个程序好坏的标准是什么？

结果正确、运行速度快、程序结构优美、算法设计合理等，这些都可以成为衡量的标准。不过，选择用什么算法是最根本的问题，它决定了程序能用还是不能用。

本章将回答以下问题：什么是算法？为什么要使用该算法？如何评价算法的好坏？如何选择算法？通过对这些问题的讲解帮助算法竞赛的初学者建立基本的计算思维。

2.1　计算的资源

程序运行时需要的资源有两种，即计算时间和存储空间。资源是有限的，一个算法对这两个资源的使用程度可以用来衡量该算法的优劣。

- 时间复杂度：程序运行需要的时间。
- 空间复杂度：程序运行需要的存储空间。

与这两个复杂度对应，OJ上的题目一般会有对运行时间和空间的说明，例如：

Time Limit：2000/1000ms(Java/Others)

Memory Limit：65 536/65 536KB(Java/Others)

Time Limit是对程序运行时间的限制，这个题目要求在2s(Java)/1s(C、C++)内结束。

Memory Limit是对程序使用内存的限制，这里是65 536KB，即64MB。

这两个限制条件非常重要，是检验程序性能的参数。不过在现场赛中，为了增加迷惑性，可能**不会列出**这两个参数，需要参赛队员自己判断。

所以，队员拿到题目后，第一步要分析的是程序运行需要的**计算时间**和**存储空间**。

编程竞赛的题目，在逻辑、数学、算法上有不同的难度：简单的，可以一眼看懂；复杂的，往往需要很多步骤才能找到解决方案。它们对程序性能考核的要求是程序必须在限定的时间和空间内运行结束。

这是因为，问题的“有效”解决不仅在于能否得到正确答案，更重要的是能在合理的时间和空间内给出答案。

李开复在“算法的力量”一文中写道：“1988年，贝尔实验室副总裁亲自来访问我的学校，目的就是想了解为什么他们的语音识别系统比我开发的慢几十倍，而且，在扩大至大词汇系统后速度差异更有几百倍之多……在与他们探讨的过程中，我惊讶地发现一个$O(nm)$

的动态规划居然被他们做成了 $O(n^2m)$……贝尔实验室的研究员当然绝顶聪明，但他们全都是学数学、物理或电机出身，从未学过计算机科学或算法，这才犯了这么基本的错误。”

上文提到的 $O(nm)$ 和 $O(n^2m)$ 就是时间复杂度。符号 O 表示复杂度，$O(nm)$ 可以粗略地理解为运行次数是 $n\times m$。$O(n^2m)$ 比 $O(nm)$ 运行时间差不多大 n 倍。

在上面这个语音识别的例子中，设 $n=100$，如果李开复的识别系统的运行时间是 1s，那么贝尔实验室的系统需要 100s。显然，一个长达 100s 才能得到结果的语音识别系统肯定是不实用的。李开复的这个例子生动地说明了“好”算法的属性——有合理的时间效率。

那么如何衡量程序运行的时间效率？测量程序在计算机上运行的时间，可以得到一个直观的认识。

下面的程序只有一个 for 语句，它对 k 进行累加，循环次数是 n。该程序用 clock()函数统计 for 循环的运行时间。

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    int i, k, n = 1e8;
    clock_t start, end;
    start = clock();
    for(i = 0; i<n; i++)  k++;          //循环次数
    end = clock();
    cout << (double)(end - start) / CLOCKS_PER_SEC << endl;
}
```

上面的程序在一台普通配置的计算机上运行，例如 CPU 为 i5-8250U、内存为 8GB、64 位的操作系统，结果如下：

当 $n=1e8=10^8$ 时，输出的运行时间是 0.164s。

当 $n=1e9$ 时，输出的运行时间是 1.645s。

评测用的 OJ 服务器，性能可能比这个好一些，也可能差不多。

所以，如果题目要求“Time Limit：2000/1000ms(Java/Others)”，那么内部的循环次数应该满足 $n\leqslant10^8$，即 1 亿次以内。

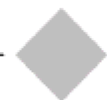
由于程序的运行时间依赖于计算机的性能，不同的计算机结果不同，所以直接把运行时间作为判断标准并不准确。通常，用程序执行的“次数”来衡量更加合理，例如上述程序循环了 n 次，把它的运行效率记为 $O(n)$。

竞赛所给的题目一般都会有多种解法，它考核的是在限定时间和空间内解决问题。如果条件很宽松，那么可以在多种解法中选一个容易编程的算法；如果给定的条件很苛刻，那么能选用的合适算法就不多了。

下面用一个例子来说明对同样的问题如何选用不同的解法。

hdu 1425 “sort”

Time Limit：6000/1000ms(Java/Others)Memory Limit：64/32MB(Java/Others)



给出 n 个整数，请按从大到小的顺序输出其中前 m 大的数。

输入：每组测试数据有两行，第 1 行有两个数 n 和 m($0<n, m<1\,000\,000$)，第 2 行包含 n 个各不相同，且都处于区间[$-500\,000, 500\,000$]的整数。

输出：对每组测试数据按从大到小的顺序输出前 m 大的数。

输入样例：

5 3

3 −35 92 213 −644

输出样例：

213 92 3

该题的思路是先对 100 万个数排序，然后输出前 m 大的数。题目给出了代码运行时间，非 Java 语言的时间是 1s，内存是 32MB。

下面分别用冒泡排序、快速排序、哈希 3 种算法编程。

1. 冒泡排序

首先用最简单的冒泡排序算法求解上面的问题。

```
#include <bits/stdc++.h>
using namespace std;
int a[1000001];                                          //记录数字
#define swap(a, b) {int temp = a; a = b; b = temp;}     //交换
int n, m;
void bubble_sort(){                                      //冒泡排序，结果仍放在 a[]中
    for(int i = 1; i <= n - 1; i++)
        for(int j = 1; j <= n - i; j++)
            if(a[j] > a[j + 1])
                swap(a[j], a[j + 1]);
}
int main(){
    while(~scanf("%d%d", &n, &m)){
        for(int i = 1; i <= n; i++)   scanf("%d", &a[i]);
        bubble_sort();
        for(int i = n; i >= n - m + 1; i -- ){           //打印前 m 大的数，反序打印
            if(i == n - m + 1)  printf("%d\n", a[i]);
            else printf("%d ", a[i]);
        }
    }
    return 0;
}
```

在 bubble_sort()运行后，得到从小到大的排序结果，然后从后往前打印前 m 大的数。冒泡排序算法的步骤如下：

(1) 第一轮，从第 1 个数到第 n 个数，逐个对比每两个相邻的数 a、b，如果 $a>b$，则交换。这一轮的结果是把最大的数“冒泡”到了第 n 个位置，在后面不用再管它。

(2) 第二轮，从第 1 个数到第 $n-1$ 个数，对比每两个相邻的数。这一轮，把第二大的数

“冒泡”到了第 $n-1$ 个位置。

(3) 继续以上过程,直到结束。

下面分析程序的时间和空间效率。

(1) 时间复杂度,也就是程序执行了多少步骤,花了多少时间。

在 bubble_sort()中有两层循环,循环次数是 $n-1+n-2+\cdots+1\approx n^2/2$;在 swap($a$,$b$)中做了 3 次操作;总的计算次数是 $3n^2/2$,复杂度记为 $O(n^2)$。当 $n=100$ 万时,计算超过 1 万亿次。如果提交到 OJ,由于 OJ 每秒只能运行 1 亿次,必然返回 TLE 超时。可以推出,只有 $n<1$ 万时才勉强能用冒泡算法。

(2) 空间复杂度,也就是程序占用的内存空间。程序使用 int a[1000001]存储数据, bubble_sort()也没有使用额外的空间。int 是 32 位整数,占用 4 个字节,所以 int a[1000001]共使用了 4MB 空间。这是冒泡算法的优点,它不额外占用空间。

2. 快速排序

快速排序是一种基于分治法的优秀排序算法。这里先直接用 STL 的 sort()函数,它是改良版的快速排序,称为“内省式排序”。

在上面的程序中,把“bubble_sort();”改为“sort(a+1, a+n+1);”就完成了 a[1]到 a[n]的排序,结果仍然保存在 a[]中。

算法的时间复杂度是 $O(n\log_2 n)$,当 $n=100$ 万时,100 万$\times\log_2$100 万$\approx$2000 万。

在 hdu 上提交,返回的运行时间是 600ms[①],正好通过 OJ 的测试。

3. 哈希算法

哈希算法是一种以空间换取时间的算法。本题的哈希思路是:在输入数字 t 的时候,在a[500000 + t]这个位置记录 a[500000 + t] = 1,在输出的时候逐个检查 a[i],如果 a[i]等于 1,表示这个数存在,打印出前 m 个数。程序如下:

```
#include<bits/stdc++.h>
using namespace std;
const int MAXN = 1000001;
int a[MAXN];
int main(){
    int n,m;
    while(~scanf("%d%d", &n, &m)){
        memset(a, 0, sizeof(a));
        for(int i=0; i<n; i++){
```

① 此题数据量很大,大量时间花在输入上,如果用 cin 输入,会 TLE,真正花在排序上的时间不多。读者可能有兴趣了解具体的执行时间,可以非常粗略地分析如下:

(a) 快排程序用了 600ms,包括输入与输出时间 A,以及排序时间 B。

(b) 哈希程序用了 500ms,它的输入与输出时间和快排程序的时间差不多,都是 A;排序时间是 C。

(c) 快排的复杂度是 $O(n\log_2 n)$,哈希的复杂度是 $O(n)$。当 $n=100$ 万时,$\log_2$100 万$\approx$20,那么 B$\approx$20C。计算得到 A$\approx$500ms,B$\approx$100ms,C$\approx$5ms。也就是说,程序的大部分时间花在了输入与输出上。

(d) 分析 $n=200$ 万的情况。快排程序的总时间$\approx$2A+B$\times$(200 万$\times\log_2$200 万)/(100 万$\times\log_2$100 万)$\approx$2A+2.1B =1210ms;哈希程序的总时间$\approx$2A+2C=1010ms。看起来似乎改善不大,因为时间大部分用在处理输入与输出上。如果一个程序的输入用时不多,那么时间就取决于排序算法。哈希算法比快排算法快 $\log_2 n$ 倍,很有优势。

```
            int t;
            scanf("%d", &t);            //此题数据多,如果用很慢的 cin 输入,肯定 TLE
            a[500000 + t] = 1;          //数字 t,登记在 500000 + t 这个位置
        }
        for(int i = MAXN; m > 0; i-- )
            if(a[i]){
                if(m > 1)  printf("%d ", i - 500000);
                else       printf("%d\n", i - 500000);
                m-- ;
            }
    }
    return 0;
}
```

程序并没有做显式的排序,只是每次把输入的数按哈希插入到对应位置,只有 1 次操作;n 个数输入完毕,就相当于排好序了。总的时间复杂度是 $O(n)$。在 hdu 上提交,返回的运行时间是 500ms。

4. 算法的选择

从上述 3 种程序可知,对于同一个问题,经常存在不同的解决方案,有高效的,也有低效的。算法编程竞赛主要的考核点就是在限定的时间和空间内解决问题。虽然在大部分情况下只有高效的算法才能通过满足判题系统的要求,但是请注意,并不是只有高效的算法才是合理的,低效的算法有时也是有用的。对于程序设计竞赛来说,由于竞赛时间极为紧张,解题速度极为关键,只有尽快完成更多的题目才能获得胜利。在满足限定条件的前提下,用最短的时间完成编码任务才是最重要的。低效算法的编码时间往往大大低于高效算法。例如,题目限定时间是 1s,现在有两个方案:①高效算法 0.01s 运行结束,但是代码有 50 行,编程 40 分钟;②低效算法 1s 运行结束,但是代码只有 20 行,编程 10 分钟。显然,此时应该选择低效算法。

不过在竞赛时,这种情况通常只发生在数据规模小的简单题中,即所谓的"签到题",而大部分题目是没有这种好事的。所以,这只是一个小小的技巧,并没有太大用处。

2.2　算法的定义

前面反复提到了"算法"这个概念,参加 ACM-ICPC、CCPC 竞赛的学生也常常说"我们在搞算法竞赛"。大家也常常听说"程序＝算法＋数据结构",算法是解决问题的逻辑、方法、过程,数据结构是数据在计算机中的存储和访问方式,二者是紧密结合的。

算法(Algorithm)是对特定问题求解步骤的一种描述,是指令的有限序列。它有以下 5 个特征。

(1) 输入:一个算法有零个或多个输入。程序可以没有输入,例如一个定时闹钟程序,它不需要输入,但是能够每隔一段时间就输出一个报警。

(2) 输出:一个算法有一个或多个输出。程序可以没有输入,但是一定要有输出。

(3) 有穷性:一个算法必须在执行有穷步之后结束,且每一步都在有穷时间内完成。

(4) 确定性：算法中的每一条指令必须有确切的含义，对于相同的输入只能得到相同的输出。

(5) 可行性：算法描述的操作可以通过已经实现的基本操作执行有限次来实现。

这里以冒泡排序算法为例，上一节已经描述过它的执行步骤，它满足上述5个特征。

(1) 输入：由 n 个数构成的序列$\{a_1, a_2, a_3, \cdots, a_n\}$。

(2) 输出：对输入的一个排序$\{a_1', a_2', a_3', \cdots, a_n'\}$，且 $a_1' \leqslant a_2' \leqslant a_3' \leqslant \cdots \leqslant a_n'$。

(3) 有穷性：算法在执行 $O(n^2)$ 次后结束，这也是对算法性能的评估，即算法复杂度。

(4) 确定性：算法的每个步骤都是确定的。

(5) 可行性：算法的步骤能编程实现。



视频讲解

需要指出的是，上述第(5)条的可行性也是很重要的。有些算法并不能编程实现，例如一个有趣的排序算法——珠排序(Bead sort[①])，如果它用重力法，能够在 $O(1)$ 或 $O(\sqrt{n})$ 时间内得到排序结果，效率高到令人惊叹，但是无法编程实现。

2.3 算法的评估

上面已经反复提到，衡量算法性能的主要标准是时间复杂度，本节再针对算法竞赛展开说明。

为什么一般不讨论空间复杂度呢？在一般情况下，一个程序的空间复杂度是容易分析的，而时间复杂度往往关系到算法的根本逻辑，更能说明一个程序的优劣。因此，如果不特别说明，在提到“复杂度”时一般指时间复杂度。

注意，时间复杂度只是一个估计，并不需要精确计算。例如，在一个有 n 个数的无序数列中查找某个数 x，可能第一个数就是 x，也可能最后一个数才是 x，平均查找时间是 $n/2$ 次，但是把查找的时间复杂度记为 $O(n)$，而不是 $O(n/2)$。再如，冒泡排序算法的计算次数约等于 $n^2/2$ 次，但是仍记为 $O(n^2)$，而不是 $O(n^2/2)$。在算法分析中，规模 n 前面的常数系数被认为是不重要的。

还有，OJ 系统所判定的运行时间是整个程序运行所花的时间，而不是理论上算法所需要的时间。同一个算法，不同的人写出的程序，复杂度和运行时间可能差别很大，跟编程语言、逻辑结构、库函数等都有关系。

一个程序或算法的复杂度有以下可能。

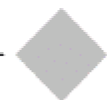
1. $O(1)$

计算时间是一个常数，和问题的规模 n 无关。例如，用公式计算时，一次计算的复杂度就是 $O(1)$；哈希算法，用 hash 函数在常数时间内计算出存储位置；在矩阵 $A[M][N]$ 中查找第 i 行第 j 列的元素，只需要访问 $A[i][j]$ 就够了。

2. $O(\log_2 n)$

计算时间是对数，通常是以2为底的对数，每一步计算后，问题的规模减小一倍。例如

① https://en.wikipedia.org/wiki/Bead_sort.



在一个长度为 n 的有序数列中查找某个数，用折半查找的方法只需要 $\log_2 n$ 次就能找到。再如分治法，一般情况下，在每个步骤把规模减小一倍，所以一共有 $O(\log_2 n)$ 个步骤。

$O(\log_2 n)$ 和 $O(1)$ 没有太大差别。

3. $O(n)$

计算时间随规模 n 线性增长。在很多情况下，这是算法可能达到的最优复杂度，因为对输入的 n 个数，程序一般需要处理所有的数，即计算 n 次。例如查找一个无序数列中的某个数，可能需要检查所有的数。再如图问题，有 V 个点和 E 个边，大多数图的问题都需要搜索到所有的点和边，复杂度的上限就是 $O(V+E)$。

4. $O(n\log_2 n)$

这常常是算法能达到的最优复杂度。例如分治法，一共 $O(\log_2 n)$ 个步骤，每个步骤对每个数操作一次，所以总复杂度是 $O(n\log_2 n)$。用分治法思想实现的快速排序算法和归并排序算法的复杂度就是 $O(n\log_2 n)$。

5. $O(n^2)$

一个两重循环的算法，复杂度是 $O(n^2)$。例如冒泡排序是典型的两重循环。类似的复杂度有 $O(n^3)$、$O(n^4)$ 等。

6. $O(2^n)$

一般对应集合问题，例如一个集合中有 n 个数，要求输出它的所有子集，子集有 2^n 个。

7. $O(n!)$

在集合问题中，如果要求按顺序输出所有的子集，那么复杂度就是 $O(n!)$。

把上面的复杂度分成两类：①多项式复杂度，包括 $O(1)$、$O(n)$、$O(n\log_2 n)$、$O(n^k)$ 等，其中 k 是一个常数；②指数复杂度，包括 $O(2^n)$、$O(n!)$ 等。

如果一个算法是多项式复杂度，称它为"高效"算法；如果一个算法是指数复杂度，则称它为"低效"算法。可以这样通俗地解释"高效"和"低效"算法的区别：多项式复杂度的算法随着规模 n 的增加可以通过堆叠硬件来实现，"砸钱"是行得通的；而指数复杂度的算法增加硬件也无济于事，其增长的速度超出了人们的想象力。

竞赛题目一般的限制时间是 1s，对应普通计算机的计算速度是每秒千万次级，那么上述的时间复杂度可以换算出能解决问题的数据规模。例如，如果一个算法的复杂度是 $O(n!)$，当 $n=11$ 时，$11!=39\ 916\ 800$，这个算法只能解决 $n\leqslant 11$ 以内的问题。

下面的表 2.1 需要牢记。

表 2.1　问题规模和可用算法

问题规模 n	可用算法的时间复杂度					
	$O(\log_2 n)$	$O(n)$	$O(n\log_2 n)$	$O(n^2)$	$O(2^n)$	$O(n!)$
$n\leqslant 11$	√	√	√	√	√	√
$n\leqslant 25$	√	√	√	√	√	×
$n\leqslant 5000$	√	√	√	√	×	×
$n\leqslant 10^6$	√	√	√	×	×	×
$n\leqslant 10^7$	√	√	×	×	×	×
$n>10^8$	√	×	×	×	×	×

第 3 章　STL 和基本数据结构

✍容器
✍队列
✍栈
✍链表
✍set
✍map

STL(Standard Template Library)是 C++的标准模板库,竞赛中很多常用的数据结构、算法在 STL 中都有,熟练地掌握它们在很多题目中能极大地简化编程。本章所介绍的内容是竞赛训练的基本内容,需要完全掌握。

STL 包含容器(container)、迭代器(iterator)、空间配置器(allocator)、配接器(adapter)、算法(algorithm)、仿函数(functor) 6 个部分。本章介绍容器和两个常用算法。

3.1　容　　器

STL 容器包括顺序式容器和关联式容器。

1. 顺序式容器

顺序式容器包括 vector、list、deque、queue、priority_queue、stack 等,它们的特点如下。

- vector:动态数组,从末尾能快速插入与删除,直接访问任何元素。
- list:双链表,从任何地方快速插入与删除。
- deque:双向队列,从前面或后面快速插入与删除,直接访问任何元素。
- queue:队列,先进先出。
- priority_queue:优先队列,最高优先级元素总是第一个出列。
- stack:栈,后进先出。

2. 关联式容器

关联式容器包括 set、multiset、map、multimap 等。

- set:集合,快速查找,不允许重复值。
- multiset:快速查找,允许重复值。
- map:一对多映射,基于关键字快速查找,不允许重复值。
- multimap:一对多映射,基于关键字快速查找,允许重复值。



3.1.1 vector


视频讲解

数组是基本的数据结构，有静态数组和动态数组两种类型。在算法竞赛中，编码的惯例是：如果空间足够，能用静态数组就用静态数组，而不用指针管理动态数组，这样编程比较简单并且不会出错；如果空间紧张，可以用STL的vector建立动态数组，不仅节约空间，而且也不易出错。

vector[①]是STL的动态数组，在运行时能根据需要改变数组大小。由于它以数组形式存储，也就是说它的内存空间是连续的，所以索引可以在常数时间内完成，但是在中间进行插入和删除操作会造成内存块的复制。另外，如果数组后面的内存空间不够，需要重新申请一块足够大的内存。这些都会影响vector的效率。

vector容器是一个模板类，能存放任何类型的对象。

1. 定义

其示例如表3.1所示。

表3.1 vector定义示例

功　　能	例　　子	说　　明
定义int型数组	vector<int> a;	默认初始化，*a*为空
	vector<int> b(a);	用*a*定义*b*
	vector<int> a(100);	*a*有100个值为0的元素
	vector<int> a(100, 6);	100个值为6的元素
定义string型数组	vector<string> a(10,"null");	10个值为null的元素
	vector<string> vec(10,"hello");	10个值为hello的元素
	vector<string> b(a.begin(), a.end());	*b*是*a*的复制
定义结构型数组	struct point { int x, y; }; vector<point> a;	*a*用来存坐标

用户还可以定义多维数组，例如定义一个二维数组：

```
vector<int> a[MAXN];
```

它的第一维大小是固定的MAXN，第二维是动态的。用这个方式可以实现图的邻接表存储，细节见本书10.2节。

2. 常用操作

vector的常用操作如表3.2所示。

表3.2 vector的常用操作

功　　能	例　　子	说　　明
赋值	a.push_back(100);	在尾部添加元素
元素个数	int size=a.size();	元素个数

① http://www.cplusplus.com/reference/vector/vector/。

续表

功　　能	例　　子	说　　明
是否为空	bool isEmpty=a.empty();	判断是否为空
打印	cout << a[0] << endl;	打印第一个元素
中间插入	a.insert(a.begin()+i, k);	在第 i 个元素前面插入 k
尾部插入	a.push_back(8);	尾部插入值为 8 的元素
尾部插入	a.insert(a.end(), 10,5);	尾部插入 10 个值为 5 的元素
删除尾部	a.pop_back();	删除末尾元素
删除区间	a.erase(a.begin()+i, a.begin()+j);	删除区间 $[i, j-1]$ 的元素
删除元素	a.erase(a.begin()+2);	删除第 3 个元素
调整大小	a.resize(n)	数组大小变为 n
清空	a.clear();	清空
翻转	reverse(a.begin(), a.end());	用函数 reverse()翻转数组
排序	sort(a.begin(), a.end());	用函数 sort()排序，从小到大排

下面用一个例题来说明 vector 的使用。

hdu 4841 “圆桌问题”

圆桌边围坐着 $2n$ 个人。其中 n 个人是好人，另外 n 个人是坏人。从第一个人开始数，数到第 m 个人，立即赶走该人；然后从被赶走的人之后开始数，再将数到的第 m 个人赶走，依此方法不断赶走围坐在圆桌边的人。

预先应如何安排这些好人与坏人的座位，才能使得在赶走 n 个人之后圆桌边围坐的剩余的 n 个人全是好人？

输入：多组数据，每组数据输入：$n,m \leqslant 32\,767$。

输出：对于每一组数据，输出 $2n$ 个大写字母，“G”表示好人，“B”表示坏人，50 个字母为一行，不允许出现空白字符。相邻数据间留有一个空行。

输入样例：

2 3

2 4

输出样例：

GBBG

BGGB

这个题目是约瑟夫问题。用 vector 模拟动态变化的圆桌，赶走 n 个人之后留下的都是好人。

程序如下：

```
#include <bits/stdc++.h>
using namespace std;
int main(){
    vector <int> table;                          //模拟圆桌
    int n, m;
```

```
    while(cin >> n >> m){
        table.clear();
        for(int i = 0; i < 2 * n; i++)  table.push_back(i); //初始化
        int pos = 0;                                        //记录当前位置
        for(int i = 0; i < n; i++){                         //赶走 n 个人
            pos = (pos + m - 1) % table.size();             //圆桌是个环,取余处理
            table.erase(table.begin() + pos);               //赶走坏人,table 人数减 1
        }
        int j = 0;
        for(int i = 0; i < 2 * n; i++){                     //打印预先安排座位
            if(!(i % 50) && i)  cout << endl;               //50 个字母一行
            if(j < table.size() && i == table[j]){          //table 留下的都是好人
                j++;
                    cout <<"G";
            }
            else
                    cout <<"B";
        }
        cout << endl << endl;                               //留一个空行
    }
    return 0;
}
```

前面提到,vector 插入或者删除中间某一项时需要线性时间,即需要把这个元素后面的所有元素往后移或往前移,复杂度是 $O(n)$。如果频繁移动,则效率很低。hdu 4841 的 vector 程序用 erase()来删除中间元素就有这个问题。

3.1.2 栈和 stack[①]

栈是基本的数据结构之一,特点是“先进后出”。例如乘坐电梯时,先进电梯的最后出来;一盒泡腾片,最先放进盒子的药片位于最底层,最后被拿出来。

头文件:#include < stack >

栈的有关操作:

```
stack < Type > s;           //定义栈,Type 为数据类型,例如 int、float、char 等
s.push(item);               //把 item 放到栈顶
s.top();                    //返回栈顶的元素,但不会删除
s.pop();                    //删除栈顶的元素,但不会返回.在出栈时需要进行两步操作,即
                            //先 top()获得栈顶元素,再 pop()删除栈顶元素
s.size();                   //返回栈中元素的个数
s.empty();                  //检查栈是否为空,如果为空,返回 true,否则返回 false
```

下面用一个例子来说明栈的应用。

hdu 1062 “Text Reverse”

翻转字符串。例如输入“olleh !dlrow”,输出“hello world!”。

① http://www.cplusplus.com/reference/stack/stack/。

用栈模拟，下面是程序：

```
#include <bits/stdc++.h>
using namespace std;
int main(){
    int n;
    char ch;
    scanf("%d",&n);  getchar();
    while(n--){
        stack<char> s;
        while(true){
            ch = getchar();                  //一次读入一个字符
            if(ch == ' '||ch == '\n'||ch == EOF){
                while(!s.empty()){
                    printf("%c",s.top());  //输出栈顶
                    s.pop();               //清除栈顶
                }
                if(ch == '\n'||ch == EOF)  break;
                printf(" ");
            }
            else  s.push(ch);                //入栈
        }
        printf("\n");
    }
return 0;
}
```

爆栈问题。栈需要用空间存储，如果深度太大，或者存进栈的数组太大，那么总数会超过系统为栈分配的空间，这样就会爆栈，即栈溢出。其解决办法有下面两种：

(1) 在程序中调大系统的栈，这种方法依赖于系统和编译器，竞赛的时候，在热身赛上可以试一试。

(2) 手工写栈。有关内容见本书 10.5 节。

【习题】

比较复杂的用到栈的例子，请练习 hdu 1237“简单计算器”，逆波兰表达式。

3.1.3 队列和 queue[①]

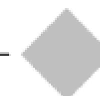
队列是基本的数据结构之一，特点是“先进先出”。例如排队，先进队列的先得到服务。

头文件：#include <queue>

队列的有关操作：

```
queue<Type> q;           //定义栈，Type 为数据类型，例如 int、float、char 等
q. push(item);           //把 item 放进队列
q.front();               //返回队首元素，但不会删除
q.pop();                 //删除队首元素
```

① http://www.cplusplus.com/reference/queue/queue/。



```
q.back();                        //返回队尾元素
q.size();                        //返回元素个数
q.empty();                       //检查队列是否为空
```

hdu 1702 “ACboy needs your help again!”

模拟栈和队列，栈是FILO，队列是FIFO。

分别用栈和队列模拟，下面是代码：

```
#include <bits/stdc++.h>
using namespace std;
int main(){
    int t,n,temp;
    cin>>t;
    while(t--){
        string str,str1;
        queue<int> Q;
        stack<int> S;
        cin>>n>>str;
        for(int i = 0; i < n; i++){
            if(str == "FIFO"){            //队列
                cin>>str1;
                if(str1 == "IN"){
                    cin>>temp;  Q.push(temp);
                }
                if(str1 == "OUT"){
                    if(Q.empty()) cout<<"None"<<endl;
                    else{
                        cout<<Q.front()<<endl;
                        Q.pop();
                    }
                }
            }
            else{                         //栈
                cin>>str1;
                if(str1 == "IN"){
                    cin>>temp;  S.push(temp);
                }
                if(str1 == "OUT"){
                    if(S.empty()) cout<<"None"<<endl;
                    else {
                        cout<<S.top()<<endl;
                        S.pop();
                    }
                }
            }
        }
    }
    return 0;
}
```

3.1.4 优先队列和 priority_queue[①]

优先队列,顾名思义就是优先级最高的先出队。它是队列和排序的完美结合,不仅可以存储数据,还可以将这些数据按照设定的规则进行排序。每次的 push 和 pop 操作,优先队列都会动态调整,把优先级最高的元素放在前面。

优先队列的有关操作如下:

```
q.top();                          //返回具有最高优先级的元素值,但不删除该元素
q.pop();                          //删除最高优先级元素
q.push(item);                     //插入新元素
```

在 STL 中,优先队列是用二叉堆来实现的,在队列中 push 一个数或 pop 一个数,复杂度都是 $O(\log_2 n)$。

可以用优先队列对数据排序:设定数据小的优先级高,把所有数 push 进优先队列后一个个 top 出来,就得到了从小到大的排序。其总复杂度是 $O(n\log_2 n)$。

图论的 Dijkstra 算法的程序实现用 STL 的优先队列能极大地简化代码,参考本书的 10.9.4 节。

【习题】

hdu 1873“看病要排队”。

3.1.5 链表和 list[②]

STL 的 list 是数据结构的双向链表,它的内存空间可以是不连续的,通过指针来进行数据的访问,它可以高效率地在任意地方删除和插入,插入和删除操作是常数时间的。

list 和 vector 的优缺点正好相反,它们的应用场景不同。

(1) vector:插入和删除操作少,随机访问元素频繁。

(2) list:插入和删除频繁,随机访问较少。

下面用一个例题来说明 list 的应用。

hdu 1276“士兵队列训练问题”

一队士兵报数:从头开始进行 1 至 2 报数,凡报到 2 的出列,剩下的向小序号方向靠拢,再从头开始进行 1 至 3 报数,凡报到 3 的出列,剩下的向小序号方向靠拢,以后从头开始轮流进行 1 至 2 报数、1 至 3 报数,直到剩下的人数不超过 3 为止。

输入:士兵人数。

输出:剩下士兵最初的编号。

① http://www.cplusplus.com/reference/queue/priority_queue/。

② http://www.cplusplus.com/reference/list/list/。

程序如下：

```
#include <bits/stdc++.h>
using namespace std;
int main(){
    int t,n;
    cin >> t;
    while(t--) {
        cin >> n;
        int k = 2;
        list<int> mylist;              //定义
        list<int>::iterator it;
        for(int i = 1;i <= n;i++)
            mylist.push_back(i);       //赋值
        while(mylist.size() > 3) {
            int num = 1;
            for(it = mylist.begin(); it != mylist.end(); ){
                if(num++ % k == 0)
                    it = mylist.erase(it);
                else
                    it++;
            }
            k == 2 ? k = 3:k = 2;      //1 至 2 报数,1 至 3 报数
        }
        for(it = mylist.begin(); it != mylist.end(); it++){
            if (it != mylist.begin())
                cout << " ";
            cout << *it;
        }
        cout << endl;
    }
    return 0;
}
```

3.1.6 set[①]

set 就是集合。STL 的 set 用二叉搜索树实现，集合中的每个元素只出现一次，并且是排好序的。访问元素的时间复杂度是 $O(\log_2 n)$，非常高效。

set 和 3.1.7 节的 map 在竞赛题中的应用很广泛，特别是需要用二叉搜索树处理数据的题目，如果用 set 或 map 实现，能极大地简化代码。

set 的有关操作：

```
set<Type> A;                    //定义
A.insert(item);                 //把 item 放进 set
A.erase(item);                  //删除元素 item
```

① http://www.cplusplus.com/reference/set/set/。

```
A.clear();                          //清空 set
A.empty ();                         //判断是否为空
A.size();                           //返回元素个数
A.find(k);                          //返回一个迭代器,指向键值 k
A.lower_bound(k);                   //返回一个迭代器,指向键值不小于 k 的第一个元素
A.upper_bound();                    //返回一个迭代器,指向键值大于 k 的第一个元素
```

下面用一个例子来说明 set 的应用。

hdu 2094 "产生冠军"

有一群人打乒乓球比赛,两两捉对厮杀,每两个人之间最多打一场比赛。

球赛的规则如下:

如果 A 打败了 B,B 又打败了 C,而 A 与 C 之间没有进行过比赛,那么就认定 A 一定能打败 C。

如果 A 打败了 B,B 又打败了 C,而且 C 又打败了 A,那么 A、B、C 三者都不可能成为冠军。

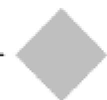
根据这个规则,无须循环较量,或许就能确定冠军。本题的任务就是对于一群比赛选手,在经过了若干场厮杀之后,确定是否已经产生了冠军。

这一题的思路是定义集合 A 和 B,把所有人放进集合 A,把所有有失败记录的放进集合 B。如果 $A-B=1$,则可以判断存在冠军,否则不能,请读者自己思考原因。

下面的程序演示了 set 的应用。

hdu 2094 程序

```
#include <bits/stdc++.h>
using namespace std;
int main(){
    set<string> A, B;                       //定义集合
    string s1, s2;
    int n;
    while(cin >> n && n){
        for(int i = 0; i < n; i++) {
            cin >> s1 >> s2;
            A.insert(s1);  A.insert(s2);    //把所有人放进集合 A
            B.insert(s2);                   //把失败者放进集合 B
        }
        if(A.size() - B.size() == 1)
            cout << "Yes" << endl;
        else
            cout << "No" << endl;
        A.clear(); B.clear();
    }
    return 0;
}
```



3.1.7　map[①]

这里有一个常见的问题：有 n 个学生，每人有姓名 name 和学号 id，现在给定一个学生的 name，要求查找他的 id。

简单的做法是定义 string name[n]和 int id[n]（可以放在一个结构体中）存储信息，然后在 name[]中查找这个学生，找到后输出他的 id。这样做的缺点是需要搜索所有的 name[]，复杂度是 $O(n)$，效率很低。

利用 STL 中的 map 容器可以快速地实现这个查找，复杂度是 $O(\log_2 n)$。

map 是关联容器，它实现从键（key）到值（value）的映射。map 效率高的原因是它用平衡二叉搜索树来存储和访问。

在上述例子中，map 的具体操作如下。

（1）定义：map <string, int> student，存储学生的 name 和 id。

（2）赋值：例如 student["Tom"]=15。这里把"Tom"当成普通数组的下标来使用。

（3）查找：在找学号时，可以直接用 student["Tom"]表示他的 id，不用再去搜索所有的姓名。

map 用起来很方便。对于它的插入、查找、访问等操作，请读者自己阅读有关资料，并且认真掌握。

下面用一个例题来简单介绍 map 的使用。

hdu 2648 "Shopping"

女孩 dandelion 经常去购物，她特别喜欢一家叫"memory"的商店。由于春节快到了，所有商店的价格每天都在上涨。她想知道这家商店每天的价格排名。

输入：

第 1 行是数字 $n(n\leqslant 10\,000)$，代表商店的数量。

后面 n 行，每行有一个字符串（长度小于 31，只包含小写字母和大写字母），表示商店的名称。

然后一行是数字 $m(1\leqslant m\leqslant 50)$，表示天数。

后面有 m 部分，每部分有 n 行，每行是数字 s 和一个字符串 p，表示商店 p 在这一天涨价 s。

输出：包含 m 行，第 i 行显示第 i 天后店铺"memory"的排名。排名的定义为如果有 t 个商店的价格高于"memory"，那么它的排名是 $t+1$。

本题代码如下：

```
#include <bits/stdc++.h>
using namespace std;
int main(){
    int n, m, p;
```

① http://www.cplusplus.com/reference/map/map/。

```
    map<string, int> shop;
    while(cin>>n) {
        string s;
        for(int i=1; i<=n; i++) cin>>s;   //输入商店名字,实际上用不着处理
        cin>>m;
        while(m--) {
            for(int i=1; i<=n; i++) {
                cin>>p>>s;
                shop[s] += p;             //用 map 可以直接操作商店,加上价格
            }
            int rank = 1;
            map<string,int>::iterator it;  //迭代器
            for(it=shop.begin(); it != shop.end(); it++)
                if(it->second > shop["memory"])    //比较价格
                    rank++;
            cout<<rank<<endl;
        }
        shop.clear();
    }
    return 0;
}
```

3.2 sort()

STL 的排序函数 sort()[①]是算法竞赛中最常用的函数之一,它的定义有以下两种:

(1) void sort(RandomAccessIterator first, RandomAccessIterator last);

(2) void sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp);

返回值:无。

复杂度:$O(n\log_2 n)$。

注意,它排序的范围是[first, last),包括 first,不包括 last。

1. sort()的比较函数

排序是对比元素的大小。sort()可以用自定义的比较函数进行排序,也可以用系统的4种函数排序,即 less()、greater()、less_equal()、greater_equal()。在默认情况下,程序是按从小到大的顺序排序的,less()可以不写。

下面是程序例子。

```
#include<bits/stdc++.h>
using namespace std;
bool my_less(int i, int j)     {return (i<j);}  //自定义小于
bool my_greater(int i, int j)  {return (i>j);}  //自定义大于
int main(){
```

① http://www.cplusplus.com/reference/algorithm/sort/。



```
    vector<int> a = {3,7,2,5,6,8,5,4};
    sort(a.begin(),a.begin()+4);                    //对前4个排序,输出 2 3 5 7 6 8 5 4
  //sort(a.begin(),a.end());                        //从小到大排序, 输出 2 3 4 5 5 6 7 8
  //sort(a.begin(),a.end(),less<int>());            //输出 2 3 4 5 5 6 7 8
  //sort(a.begin(),a.end(),my_less);                //自定义排序,输出 2 3 4 5 5 6 7 8
  //sort(a.begin(),a.end(),greater<int>());         //从大到小排序,输出 8 7 6 5 5 4 3 2
  //sort(a.begin(),a.end(),my_greater);             //输出 8 7 6 5 5 4 3 2
    for(int i=0; i<a.size(); i++)                   //输出
        cout << a[i]<< " ";
    return 0;
}
```

sort()还可以对结构变量进行排序,例如:

```
struct Student{
    char name[256];
    int score;
};
bool compare(struct Student * a,struct Student * b){   //按分数从大到小排序
    return a->score > b->score;
}
…
vector<struct Student *> list;                         //定义 list,把学生信息存到 list 里
…
sort(list.begin(), list.end(), compare);               //按分数排序
```

2. 相关函数

stable_sort():当排序元素相等时,保留原来的顺序。在对结构体排序时,当结构体中的排序元素相等时,如果需要保留原序,可以用 stable_sort()。

partial_sort():局部排序。例如有 10 个数字,求最小的 5 个数。如果用 sort(),需要先全部排序,再输出前 5 个;而用 partial_sort()可以直接输出前 5 个。

3.3　next_permutation()

STL 提供求下一个排列组合的函数 next_permutation()[①]。例如 3 个字符 a、b、c 组成的序列,next_permutation()能按字典序返回 6 个组合,即 abc、acb、bac、bca、cab、cba。

函数 next_permutation()的定义有下面两种形式:

(1) bool next_permutation(BidirectionalIterator first, BidirectionalIterator last);

(2) bool next_permutation(BidirectionalIterator first, BidirectionalIterator last, Compare comp);

返回值:如果没有下一个排列组合,返回 false,否则返回 true。每执行 next_permutation()

① http://www.cplusplus.com/reference/algorithm/next_permutation/。

一次，就会把新的排列放到原来的空间里。

复杂度：$O(n)$。

注意，它排列的范围是[first, last)，包括 first，不包括 last。

在使用 next_permutation()的时候，初始序列一般是一个字典序最小的序列，如果不是，可以用 sort()排序，得到最小序列，然后再使用 next_permutation()，例题见本书的 4.1 节。

下面的例题是该函数的一个简单应用。

hdu 1027 "Ignatius and the Princess II"

给定 n 个数字，从 1 到 n，要求输出第 m 小的序列。

输入：数字 n 和 m，$1 \leqslant n \leqslant 1000$，$1 \leqslant m \leqslant 10\,000$。

输出：输出第 m 小的序列。

程序的思路是首先生成一个 $123\cdots n$ 的最小字典序列，即初始序列，然后用 next_permutation()一个一个地生成下一个字典序更大的序列。

程序如下：

```
#include <bits/stdc++.h>
using namespace std;
int a[1001];
int main(){
    int n, m;
    while(cin >> n >> m){
        for(int i = 1; i <= n; i++) a[i] = i;     //生成一个字典序最小的序列
        int b = 1;
        do{
            if(b == m) break;
            b++;
        }while(next_permutation(a + 1,a + n + 1));
                                                  //注意第一个是 a + 1,最后一个是 a + n
        for(int i = 1; i < n; i++)                //输出第 m 大的字典序
            cout << a[i] << " ";
        cout << a[n] << endl;
    }
    return 0;
}
```

与 next_permutation()相关的函数如下：

- prev_permutation()：求前一个排列组合。
- lexicographical_compare()：字典比较。

【习题】

hdu 1716 "排列 2"。

第4章 搜索技术

✍递归和排列
✍子集生成和组合问题
✍BFS和队列
✍A*算法
✍DFS和递归
✍八数码问题
✍回溯与剪枝
✍迭代加深搜索
✍IDA*

搜索是基本的编程技术，在算法竞赛学习中是基础的基础。搜索使用的算法是BFS和DFS，BFS用队列、DFS用递归来具体实现。在BFS和DFS的基础上可以扩展出A*算法、双向广搜算法、迭代加深搜索、IDA*等技术。本章详细介绍了这些知识点。

搜索技术是“暴力法”算法思想的具体实现。

人们常说：“要利用计算机强大的计算能力。”如果答案在一大堆数字里面，让计算机一个个去试，符合条件的不就是答案了吗？

没错，最基本的算法思想“暴力法”就是这样做的。例如，银行卡密码是6位数字，共100万个，对于计算机来说，尝试100万次只需要一瞬间。不过计算机也不是无敌的。为了应对计算机强大的计算能力，可以对密码进行强化设计。例如网络账号密码，大部分网站都要求长度在8位以上，并且混合数字、字母、标点等。从40多个符号中选8个组成密码，数量有40×39×38×37×36×35×34×33>3万亿，即使用计算机也不能很快算出来。

暴力法(Brute force，又译为蛮力法)：把所有可能的情况都罗列出来，然后逐一检查，从中找到答案。这种方法简单、直接，不玩花样，利用了计算机强大的计算能力。

虽然暴力法常常是低效的代名词，但是它仍然很有用，原因如下：

(1) 很多问题只能用暴力法解决，例如猜密码。

(2) 对于小规模的问题，暴力法完全够用，而且避免了高级算法需要的复杂编码，在竞赛中可以加快解题速度。在竞赛中也可以用暴力法来构造测试数据，以验证高级算法的正确性。

(3) 把暴力法当作参照(benchmark)。既然暴力法是“最差”的，那么可以把它当成一个比较来衡量另外的算法有多“好”。拿到题目后，如果没有其他思路，可以先试试暴力法，看是否能帮助产生灵感。

不过，在具体编程时常常需要对暴力法进行优化，以减少搜索空间，提高效率。例如利用剪枝技术跳过不符合要求的情况，从而减少复杂度。

虽然暴力搜索的思路很简单,但是操作起来并不容易。一般有以下操作:

(1) 找到所有可能的数据,并且用数据结构表示和存储。

(2) 剪枝。尽量多地排除不符合条件的数据,以减少搜索的空间。

(3) 用某个算法快速检索这些数据。

其中的第一步就可能很不容易。例如迷宫问题,如何列举从起点到终点的所有可能的路径[①]? 再如图论中的“最短路径问题”,在地图上任取两个点,它们之间所有可行的路径可能是天文数字,以至于根本不能一一列举出来。所以计算最短路径的 Dijkstra 算法是用贪心法,进行从局部扩散到全局的搜索,不用列举所有可能的路径。

暴力法的主要操作是搜索,搜索的主要技术是 BFS 和 DFS。掌握搜索技术是学习算法竞赛的基础。在搜索时,具体的问题会有相应的数据结构,例如队列、栈、图、树等,读者应该能熟练地在这些数据结构上进行搜索的操作。

本章主要讲解 BFS 和 DFS,以及基于它们的优化技术,并以一些经典的搜索问题为例讲解算法思想,例如排列组合、生成子集、八皇后、八数码、图遍历等。

4.1 递归和排列

排列和组合问题是在暴力枚举的时候经常遇到的,一般有 3 种常见情况。

问题 4.1:打印 n 个数的全排列,共 $n!$ 个。

问题 4.2:打印 n 个数中任意 m 个数的全排列,共$\frac{n!}{(n-m)!}$个。

问题 4.3:打印 n 个数中任意 m 个数的组合,共 $C_n^m=\frac{n!}{m!(n-m)!}$个。

本节用递归程序来实现问题 4.1 和问题 4.2,问题 4.3 将在下一节中讲解。

在计算机编程教材中都会提到递归的概念和应用,一般会用数学中的递推方程来讲解递归的概念,例如 $f(n)=f(n-1)+f(n-2)$。在计算机系统中,递归是通过嵌套来实现的,涉及指针、地址、栈的使用。

从算法思想上看,递归是把大问题逐步缩小,直到变成最小的同类问题的过程。例如 $n\to n-1\to n-2\to\cdots\to 1$,最后的小问题的解是已知的,一般是给定的初始条件。在递归的过程中,由于大问题和小问题的解决方法完全一样,那么大家自然可以想到,大问题的程序和小问题的程序可以写成一样。一个递归函数直接调用自己,就实现了程序的复用。

递归和分治法的思路非常相似,分治是把一个大问题分解为多个类型相同的子问题。事实上,一些涉及分治法的问题可以用递归来编程,典型的有快速排序、归并排序等。

对于编程初学者来说,递归是一个难以理解的编程概念,很容易绕晕。为了帮助理解,可以一步步打印出递归函数的输出,看它从大到小解决问题的过程。

编程竞赛中的暴力法常常需要考虑所有可能的情况,用递归编程可以轻松、方便地实现对搜索空间所有状态的遍历。

---

① 用 DFS 可以实现,程序也非常短。在学完本章之后,读者就能轻松地写出程序。

【问题 4.1】　打印 n 个数的全排列。

在用递归解决这个问题之前先给出 STL 的实现方法。

1. 用 STL 输出全排列

如果需要全排列的场景比较简单，可以直接用 C++ STL 的库函数 next_permutation()，它按字典序输出下一个排列。在使用之前，先用 sort()给数据排序，得到最小排列，然后每调用 next_permutation()一次，就得到一个大一点的排列。

next_permutation()的优点是能按从小到大的顺序输出排列。

```
#include <iostream>
#include <algorithm>                          //包含 sort()和 next_permutation()函数
using namespace std;
int main(){
    int data[4] = {5, 2, 1, 4};
    sort(data, data + 4);                     //排序,得到最小排列
    do{
        for(int i = 0; i < 4; ++i)            //输出一个排列
            cout << data[i] << " ";
        cout << endl;
    }while(next_permutation(data, data + 4)); //把下一个排列放在 data 中
    return 0;
}
```

2. 用递归求全排列

下面用递归求全排列，代码很短，但是理解起来并不容易。读者可以自己打印每一个全排列的输出，然后认真理解。



视频讲解

在用递归之前，为了对比，先给出一个简单、粗暴的方法：以 10 个数的全排列为例，用排列组合的思路写一个 10 级的 for 循环，在每个 for 中选一个和前面的 for 用过的都不同的数。当 $n=10$ 时，一共有 $10!=3\ 628\ 800$ 个排列。

```
#include <bits/stdc++.h>
using namespace std;
int data[] = {7,1,2,3,4,5,6,8,9,10,12};         //本例子中用到前 10 个数
int main(){
    int num = 10;
    int i, j, k, m, n, p, q, r, s, t;           //10 个 for 循环
    for(i = 0; i < num; i++)
          for(j = 0; j < num; j++)
              if(j != i)                        //让 j 不等于 i
              for(k = 0; k < num; k++)
                if(k != j && k != i)            //让 k 不等于 i、j
                    for(m = 0; m < num; m++)
                        if(m!= j && m!= i && m!= k)   //让 m 不等于 i、j、k
                            ...
                //最后打印出一个全排列: cout << data[i]<< data[j]...
}
```

上述的程序看起来很“笨”，下面用递归来写，显得很“美”。

用递归求全排列的思路：设定数字是{1 2 3 4 5…n}

(1) 让第1个数不同，得到n个数列。其办法是把第1个和后面的每个数交换。

1 2 3 4 5…n

2 1 3 4 5…n

⋮

n 2 3 4 5…1

以上n个数列，只要第1个数不同，不管后面的$n-1$个数是怎么排列的，这n个数列都不同。

这是递归的第一层。

(2) 继续：在上面的每个数列中去掉第1个数，对后面的$n-1$个数进行类似的排列。例如从上面第2行的{2 1 3 4 5…n}进入第二层(去掉首位2)：

1 3 4 5…n

3 1 4 5…n

⋮

n 3 4 5…1

以上$n-1$个数列，只要第1个数不同，不管后面的$n-2$个数是怎么排列的，这$n-1$个数列都不同。

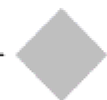
这是递归的第二层。

(3) 重复以上步骤，直到用完所有数字。

在上面所有过程完成后，数列的总个数是$n\times(n-1)\times(n-2)\cdots\times1=n!$。

递归打印全排列

```
#include<bits/stdc++.h>
using namespace std;
#define Swap(a, b) {int temp = a; a = b; b = temp;}
            //交换,也可以直接用 C++ STL 中的 swap()函数,但是速度慢一些
int data[] = {1,2,3,4,5,6,8,9,10,32,15,18,33};   //本例子中只用到前面 10 个数
int num = 0;                                      //统计全排列的个数,验证是不是 3628800
int Perm(int begin, int end){
    int i;
    if(begin == end) {                            //递归结束,产生一个全排列
                                                  //如果有必要,在此打印或处理这个全排列
        num++;                                    //统计全排列的个数
    }
    else
        for(i = begin; i <= end; i++) {
            Swap(data[begin], data[i]);           //把当前第 1 个数与后面的所有数交换位置
            Perm(begin + 1, end);
            Swap(data[begin], data[i]);           //恢复,用于下一次交换
        }
}
int main(){
    Perm(0, 9);                                   //求 10 个数的全排列
```



```
        cout << num << endl;                    //打印出排列总数,num = 10! = 3628800
    }
```

用这个程序可以检验普通计算机的计算能力。在上面的程序中加入 clock()统计时间:

```
#include <ctime>
int main() {
    clock_t start, end;
    start = clock();
    Perm(0, 9);
    end = clock();
    cout << (double)(end - start) / CLOCKS_PER_SEC << endl;
}
```

在作者的笔记本电脑上运行上述程序:

(1) Perm(0,9),计算 10 个数的全排列:10!=3 628 800,用时 0.055s。

(2) Perm(0,10),计算 11 个数的全排列:11!=39 916 800,用时 0.598s。

(3) Perm(0,11),计算 12 个数的全排列:12!=479 001 600,用时 7.305s。

12!/11!/10!的比值与 7.305s/0.598s/0.055s 的比值非常接近。

结论:笔记本电脑的计算能力大约是每秒千万次数量级。

竞赛题在一般情况下限时 1s,所以对于需要全排列的题目,其元素个数应该少于 11 个。

需要注意的是,从算法复杂度上看,上述两个程序的复杂度一样,都是 $O(n!)$。对于求全排列这样的问题,不可能有复杂度小于 $O(n!)$的算法,因为输出的数量就是 $n!$。在算法理论中,对必须要输出的元素进行的计数叫作“平凡下界”,这是程序运行所需要的最少花费。

上面的程序只要进行小的修改就能解决问题 4.2。

【问题 4.2】 打印 n 个数中任意 m 个数的全排列。

例如在 10 个数中取任意 3 个数的全排列,在 Perm()中只修改一个地方就可以了:

```
if(begin == 3) {                              //把 Perm()函数中的 end 改为 3 即可,其他都不变
    cout << data[0]<< data[1]<< data[2]<< endl;//打印 10 个数中 3 个数的全排列
    num++;                                    //统计全排列的个数,应该是 10×9×8 = 720 个
}
```

【问题 4.3】 打印 n 个数中任意 m 个数的组合。

问题 4.3 和问题 4.1 的区别为排列是有序的,组合是无序的。其中一个特例是在 n 个数中取 n 个数的组合,只有 1 种情况,就是这 n 个数本身。

问题 4.3 将在 4.2 节中讲解。

4.2　子集生成和组合问题

在 4.1 节求 10 个数的排列问题中,如果不需要输出全排列,而是输出组合,即子集(子集内部的元素是没有顺序的),那么该如何做呢?

一个包含 n 个元素的集合$\{a_0, a_1, a_2, a_3, \cdots, a_{n-1}\}$,它的子集有$\{\phi\}$,$\{a_0\}$,$\{a_1\}$,$\{a_2\}$,…,$\{a_0, a_1, a_2\}$,…,$\{a_0, a_1, a_2, a_3, \cdots, a_{n-1}\}$,共 2^n 个。

用二进制的概念进行对照是最直观的。

例如 $n=3$ 的集合$\{a_0, a_1, a_2\}$,它的子集和二进制数的对应关系如表 4.1 所示。

表 4.1　$n=3$ 的集合$\{a_0, a_1, a_2\}$的子集和二进制数的对应关系

子　　集	ϕ	a_0	a_1	a_1, a_0	a_2	a_2, a_0	a_2, a_1	a_2, a_1, a_0
二进制数	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1

所以,每个子集对应一个二进制数,这个二进制数中的每个 1 都对应着这个子集中的某个元素,而且子集中的元素是没有顺序的。

从这个表也可以理解为什么子集的数量是 2^n 个,因为所有二进制数的总个数是 2^n。

下面的程序通过处理每个二进制数中的 1 打印出了所有的子集。

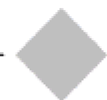
```
#include<bits/stdc++.h>
using namespace std;
void print_subset(int n){
    for(int i=0;i<(1<<n);i++) {
      //i:0~2^n,每个 i 的二进制数对应一个子集,一次打印一个子集,最后得到所有子集
        for(int j=0;j<n;j++)          //打印一个子集,即打印 i 的二进制数中所有的 1
            if(i & (1<<j))            //从 i 的最低位开始逐个检查每一位,如果是 1,打印
                cout<<j<<" ";
        cout<<endl;
    }
}
int main(){
    int n;
    cin>>n;                           //n:集合中元素的总数量
    print_subset(n);                  //打印所有的子集
}
```

回到问题 4.3:打印 n 个数中任意 m 个数的组合。对照子集生成的二进制方法,已经知道一个子集对应一个二进制数。那么一个有 k 个元素的子集,它对应的二进制数中有 k 个 1。所以,问题就转化为查找 1 的个数为 k 的二进制数,这些二进制数就是需要打印的子集。

那么如何判断二进制数中 1 的个数为 k[①]? 简单的方法是对这个 n 位二进制数逐位检查,共需要检查 n 次。

另外有一个更快的方法,它可以直接定位二进制数中 1 的位置,跳过中间的 0。它用到一个神奇的操作——$kk=kk \ \& \ (kk-1)$,功能是消除 kk 的二进制数的最后一个 1。连续进行这个操作,每次消除一个 1,直到全部消除为止,操作次数就是 1 的个数。例如二进制数 1011,经过连续 3 次操作后,所有的 1 都消除了:

① glibc 有处理二进制数的内部函数,其中 int __builtin_popcount(unsigned int x)直接返回 x 中 1 的个数。



$$1011\ \&\ (1011-1)=1011\ \&\ 1010=1010$$
$$1010\ \&\ (1010-1)=1010\ \&\ 1001=1000$$
$$1000\ \&\ (1000-1)=1000\ \&\ 0111=0000$$

利用这个操作可以计算出二进制数中 1 的个数。用 num 统计 1 的个数，具体步骤如下：

(1) 用 $kk=kk\ \&\ (kk-1)$清除 kk 的最后一个 1。

(2) num++。

(3) 继续上述操作，直到 $kk=0$。

在树状数组中也有一个类似的操作——lowbit(x)$=x\ \&\ -x$，功能是计算 x 的二进制数的最后一个 1。

下面的程序在子集生成程序的基础上实现了问题 4.3 的要求：

```
#include<bits/stdc++.h>
using namespace std;
void print_set(int n,int k){
    for(int i = 0; i<(1<<n); i++){
        int num = 0, kk = i;          //num 统计 i 中 1 的个数;kk 用来处理 i
        while(kk){
            kk = kk&(kk-1);           //清除 kk 中的最后一个 1
            num++;                    //统计 1 的个数
        }
        if(num == k){                 //二进制数中的 1 有 k 个,符合条件
            for(int j = 0; j<n; j++)
                if(i & (1<<j))
                    cout << j << " ";
            cout << endl;
        }
    }
}
int main(){
    int n, k;                         //n:元素的总数量; k:个数为 k 的子集
    cin >> n >> k;
    print_set(n,k);
}
```

4.3　BFS

4.3.1　BFS 和队列

深度优先搜索(Depth-First Search，DFS)和广度优先搜索(Breadth-First Search，BFS，或称为宽度优先搜索)是基本的暴力技术，常用于解决图、树的遍历问题。

首先考虑算法思路。以老鼠走迷宫为例，这是 DFS 和 BFS 在现实中的模型。迷宫内部的路错综复杂，老鼠从入口进去后怎么才能找到出口？有两种不同的方法：

(1) 一只老鼠走迷宫。它在每个路口都选择先走右边(当然,选择先走左边也可以),能走多远就走多远,直到碰壁无法继续往前走,然后回退一步,这一次走左边,接着继续往下走。用这个办法能走遍**所有**的路,而且**不会重复**(这里规定回退不算重复走)。这个思路就是 DFS。

(2) 一群老鼠走迷宫。假设老鼠是无限多的,这群老鼠进去后,在每个路口派出部分老鼠探索所有没走过的路。走某条路的老鼠,如果碰壁无法前行,就停下;如果到达的路口已经有其他老鼠探索过了,也停下。很显然,**所有**的道路都会走到,而且**不会重复**。这个思路就是 BFS。BFS 看起来像"并行计算",不过,由于程序是单机顺序运行的,所以可以把 BFS 看成是并行计算的模拟。

在具体编程时,一般用队列这种数据结构来具体实现 BFS,甚至可以说"BFS=队列";对于 DFS,也可以说"DFS=递归",因为用递归实现 DFS 是最普遍的。DFS 也可以用"栈"这种数据结构来直接实现,栈和递归在算法思想上是一致的。

下面用一个图遍历的题目来介绍 BFS 和队列。

hdu 1312 "Red and Black"

有一个长方形的房间,铺着方形瓷砖,瓷砖为红色或黑色。一个人站在黑色瓷砖上,他可以按上、下、左、右方向移动到相邻的瓷砖。但他不能在红色瓷砖上移动,只能在黑色瓷砖上移动。编程计算他可以到达的黑色瓷砖的数量。

输入:第 1 行包含两个正整数 W 和 H,W 和 H 分别表示 x 方向和 y 方向上的瓷砖数量。W 和 H 均不超过 20。下面有 H 行,每行包含 W 个字符。每个字符表示一片瓷砖的颜色。用符号表示如下:"•"表示黑色瓷砖;"#"表示红色瓷砖;"@"代表黑色瓷砖上的人,在数据集中只出现一次。

输出:一个数字,这个人从初始瓷砖能到达的瓷砖总数量(包括起点)。

这个题目跟老鼠走迷宫差不多:"#"相当于不能走的陷阱或墙壁,"•"是可以走的路。下面按"一群老鼠走迷宫"的思路编程。

要遍历所有可能的点,可以这样走:从起点 1 出发,走到它所有的邻居 2、3;逐一处理每个邻居,例如在邻居 2 上,再走它的所有邻居 4、5、6;继续以上过程,直到所有点都被走到,如图 4.1 所示。这是一个"扩散"的过程,如果把搜索空间看成一个池塘,丢一颗石头到起点位置,激起的波浪会一层层扩散到整个空间。需要注意的是,扩散按从近到远的顺序进行,因此,从每个被扩散到的点到起点的路径都是最短的。这个特征对解决迷宫这样的最短路径问题很有用。

用队列来处理这个扩散过程非常清晰、易懂,对照图 4.1:

(a) 1 进队。当前队列是{1}。

(b) 1 出队,1 的邻居 2、3 进队。当前队列是{2,3}(可以理解为从 1 扩散到 2、3)。

(c) 2 出队,2 的邻居 4、5、6 进队。当前队列是{3,4,5,6}(可以理解为从 2 扩散到 4、5、6)。

(d) 3 出队,7、8 进队。当前队列是{4,5,6,7,8}(可以理解为从 3 扩散到 7、8)。

(e) 4 出队,9 进队。当前队列是{5,6,7,8,9}。

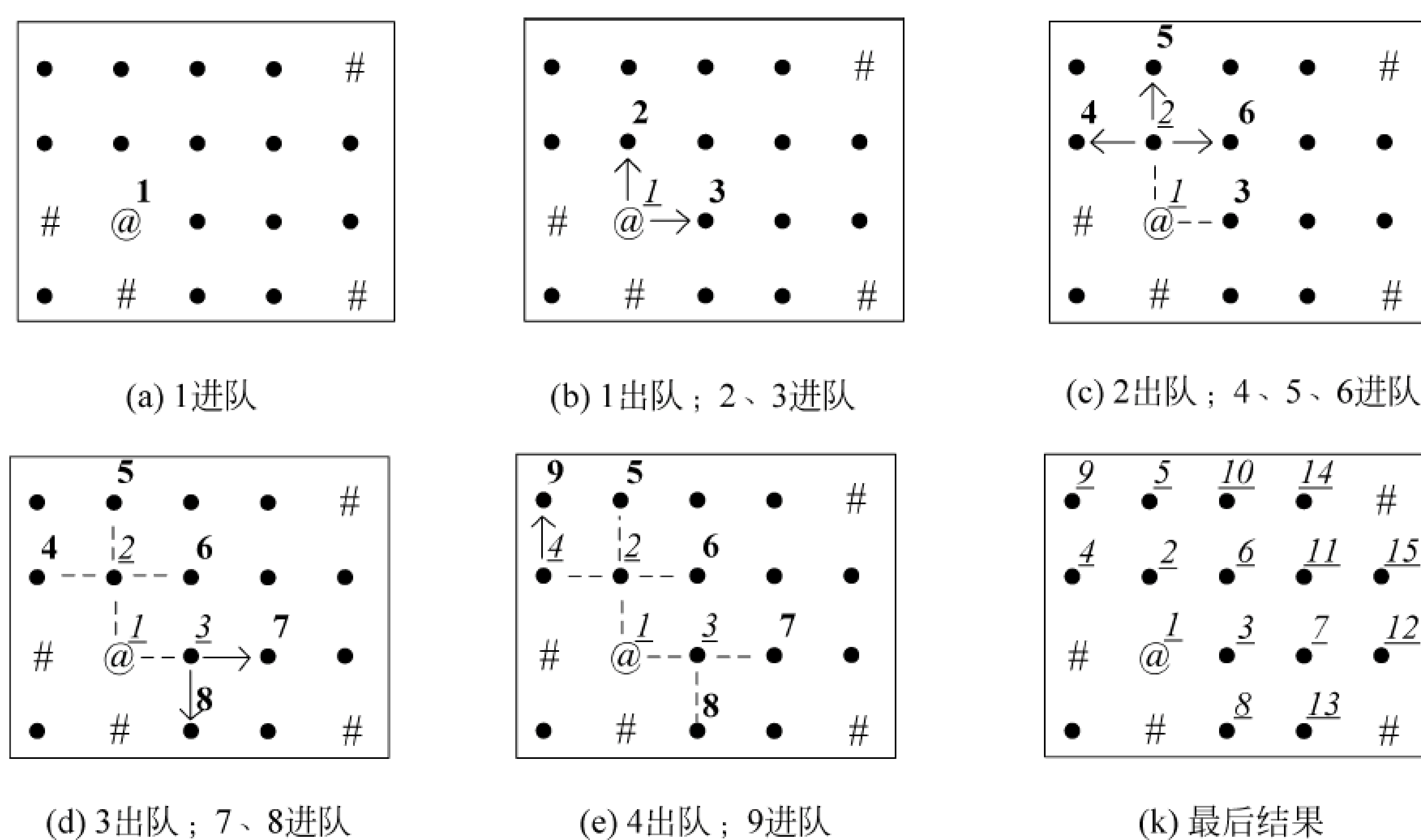


图 4.1　BFS 过程

(f) 5 出队,10 进队。当前队列是{6,7,8,9,10}。

(g) 6 出队,11 进队。当前队列是{7,8,9,10,11}。

(h) 7 出队,12、13 进队。当前队列是{8,9,10,11,12,13}。

(i) 8、9 出队,10 出队,14 进队。当前队列是{11,12,13,14}。

(j) 11 出队,15 进队。当前队列是{12,13,14,15}。

(k) 12、13、14、15 出队。当前队列是空{},结束。

hdu 1312 题的 BFS 程序

```
#include <bits/stdc++.h>
using namespace std;
char room[23][23];
int dir[4][2] = {
    {-1,0},                          //向左.左上角的坐标是(0, 0)
    {0, -1},                         //向上
    {1,0},                           //向右
    {0,1}                            //向下
};
int Wx, Hy, num;                     //Wx 行,Hy 列.用 num 统计可走的位置有多少
#define CHECK(x, y) (x < Wx && x >= 0 && y >= 0 && y < Hy)      //是否在 room 中
struct node {int x,y;};
void BFS(int dx,int dy){
    num = 1;                         //起点也包含在砖块内
    queue < node > q;                //队列中放坐标点
    node start, next;
    start.x = dx;
    start.y = dy;
    q.push(start);
    while(!q.empty()) {
        start = q.front();
```

```
            q.pop();
            //cout <<"out"<< start.x << start.y << endl;   //打印出队列情况,进行验证
            for(int i = 0; i < 4; i++) {                   //按左、上、右、下 4 个方向顺时针逐一搜索
                next.x = start.x + dir[i][0];
                next.y = start.y + dir[i][1];
                if(CHECK(next.x,next.y) && room[next.x][next.y] == '.') {
                    room[next.x][next.y] = '#';             //进队之后标记为已经处理过
                    num++;
                    q.push(next);
                }
            }
        }
}
int main(){
    int x, y, dx, dy;
    while (cin >> Wx >> Hy) {                               //Wx 行,Hy 列
        if (Wx == 0 && Hy == 0)                             //结束
            break;
        for (y = 0; y < Hy; y++) {                          //有 Hy 列
            for (x = 0; x < Wx; x++) {                      //一次读入一行
                cin >> room[x][y];
                if(room[x][y] == '@') {                     //读入起点
                    dx = x;
                    dy = y;
                }
            }
        }
        num = 0;
        BFS(dx, dy);
        cout << num << endl;
    }
    return 0;
}
```

【习题】


视频讲解

poj 3278 "Catch That Cow"。

poj 1426 "Find The Multiple"。

poj 3126 "Prime Path"。

poj 3414 "Pots"。

hdu 1240 "Asteroids!"。

hdu 4460 "Friend Chains"。

4.3.2　八数码问题和状态图搜索

BFS 搜索处理的对象不仅可以是一个数,还可以是一种"状态"。八数码问题是典型的状态图搜索问题。

1. 八数码问题

在一个 3×3 的棋盘上放置编号为 1～8 的 8 个方块，每个占一格，另外还有一个空格。与空格相邻的数字方块可以移动到空格里。任务 1：指定初始棋局和目标棋局(如图 4.2 所示)，计算出最少的移动步数；任务 2：输出数码的移动序列。

把空格看成 0，一共有 9 个数字。

输入样例：

1 2 3 0 8 4 7 6 5

1 0 3 8 2 4 7 6 5

输出样例：

2

1	2	3
	8	4
7	6	5

1		3
8	2	4
7	6	5

图 4.2　初始棋局和目标棋局

把一个棋局看成一个状态图，总共有 9!=362 880 个状态。从初始棋局开始，每次移动转到下一个状态，到达目标棋局后停止。

八数码问题是一个经典的 BFS 问题。前面章节中提到 BFS 是从近到远的扩散过程，适合解决最短距离问题。八数码从初始状态出发，每次转移都逐步逼近目标状态。每转移一次，步数加一，当到达目标时，经过的步数就是最短路径。

图 4.3 是样例的转移过程。该图中起点为(*A*，0)，*A* 表示状态，即{1 2 3 0 8 4 7 6 5}这个棋局；0 是距离起点的步数。从初始状态 *A* 出发，移动数字 0 到邻居位置，按左、上、右、下的顺时针顺序，有 3 个转移状态 *B*、*C*、*D*；目标状态是 *F*，停止。

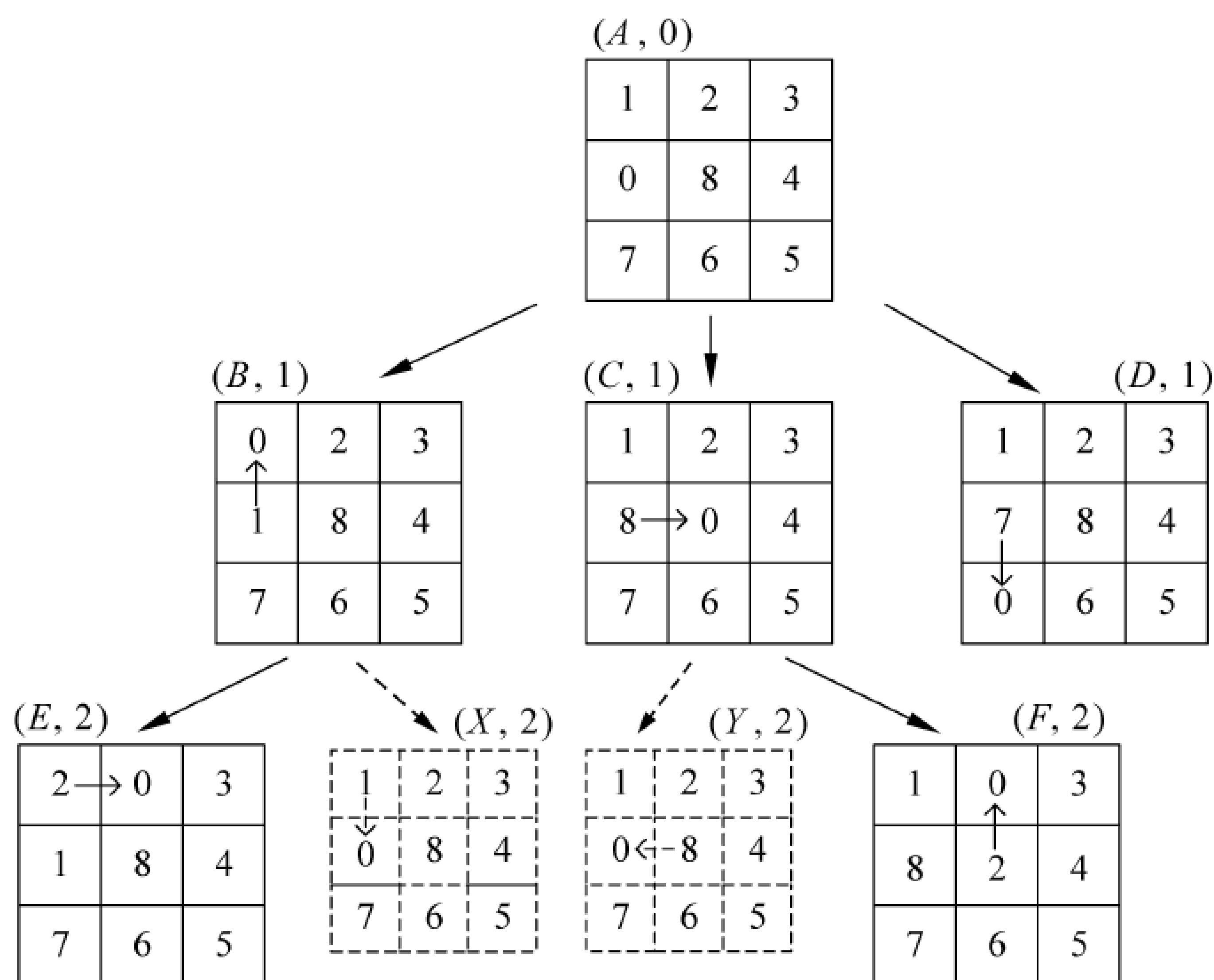


图 4.3　八数码问题的搜索树

用队列描述这个 BFS 过程：

(1) *A* 进队，当前队列是{*A*}；

(2) *A* 出队，*A* 的邻居 *B*、*C*、*D* 进队，当前队列是{*B*，*C*，*D*}，步数为 1；

(3) *B* 出队，*E* 进队，当前队列是{*C*，*D*，*E*}，*E* 的步数为 2；

(4) *C* 出队，转移到 *F*，检验 *F* 是目标状态，停止，输出 *F* 的步数 2。

仔细分析上述过程，发现从 B 状态出发实际上有 E、X 两个转移方向，而 X 正好是初始状态 A，重复了。同理 Y 状态也是重复的。如果不去掉这些重复的状态，程序会产生很多无效操作，复杂度大大增加。因此，八数码的重要问题其实是判重。

如果用暴力的方法判重，每次把新状态与 9！＝362 880 个状态对比，可能有 9！×9！次检查，不可行。因此需要一个快速的判重方法。

本题可以用数学方法“康托展开（Cantor Expansion）”来判重。

2. 康托展开

康托展开是一种特殊的哈希函数。在本题中，康托展开完成了如表 4.2 所示的工作：

表 4.2　本题中康托展开完成的工作

状　　态	012345678	012345687	012345768	012345786	…	876543210
Cantor	0	1	2	3	…	362 880－1

第 1 行是 0～8 这 9 个数字的全排列，共 9！＝362 880 个，按从小到大排序。第 2 行是每个排列对应的位置，例如最小的{012345678}在第 0 个位置，最大的{876543210}在最后的 362 880－1 这个位置。

函数 Cantor()实现的功能是：输入一个排列，即第 1 行的某个排列，计算出它的 Cantor 值，即第 2 行对应的数。

Cantor()的复杂度为 $O(n^2)$，n 是集合中元素的个数。在本题中，完成搜索和判重的总复杂度是 $O(n!n^2)$，远比用暴力判重的总复杂度 $O(n!n!)$小。

有了这个函数，八数码的程序能很快判重：每转移到一个新状态，就用 Cantor()判断这个状态是否处理过，如果处理过，则不转移。

下面举例讲解康托展开的**原理**。

例子：判断 2143 是{1，2，3，4}的全排列中第几大的数。

计算排在 2143 前面的排列数目，可以将问题转换为以下排列的和：

（1）首位小于 2 的所有排列。比 2 小的只有 1 一个数，后面 3 个数的排列有 3×2×1＝3！个（即 1234、1243、1324、1342、1423、1432），写成 1×3！＝6。

（2）首位为 2、第 2 位小于 1 的所有排列。无，写成 0×2！＝0。

（3）前两位为 21、第 3 位小于 4 的所有排列。只有 3 一个数（即 2134），写成 1×1！＝1。

（4）前 3 位为 214、第 4 位小于 3 的所有排列。无，写成 0×0！＝0。

求和：1×3！＋0×2！＋1×1！＋0×0！＝7，所以 2143 是第 8 大的数。如果用 int visited[24]数组记录各排列的位置，{2143}就是 visited[7]；第一次访问这个排列时，置 visited[7]＝1；当再次访问这个排列的时候发现 visited[7]等于 1，说明已经处理过，判重。

根据上面的例子得到康托展开公式。

把一个集合产生的全排列按字典序排序，第 X 个排列的计算公式如下：

$$X=a[n]\times(n-1)!+a[n-1]\times(n-2)!+\cdots+a[i]\times(i-1)!+\cdots+a[2]\times1!+a[1]\times0![1]$$

其中，$a[i]$为当前未出现的元素排在第几个（从 0 开始），并且有 $0\leqslant a[i]<i\ (1\leqslant i\leqslant n)$。

上述过程的反过程是康托逆展开：某个集合的全排列，输入一个数字 k，返回第 k 大的

排列。

下面的程序用“BFS+Cantor”解决了八数码问题，其中 BFS 用 STL 的 queue 实现[①]。

```
#include <bits/stdc++.h>
const int LEN = 362880;          //状态共 9!= 362 880 种
using namespace std;
struct node{
    int state[9];                //记录一个八数码的排列，即一个状态
    int dis;                     //记录到起点的距离
};

int dir[4][2] = {{-1,0}, {0,-1},{1,0},{0,1}};
                                 //左、上、右、下顺时针方向.左上角的坐标是(0,0)
int visited[LEN] = {0};          //与每个状态对应的记录，Cantor()函数对它置数，并判重
int start[9];                    //开始状态
int goal[9];                     //目标状态
long int factory[] = {1,1,2,6,24,120,720,5040,40320,362880};
                                 //Cantor()用到的常数
bool Cantor(int str[], int n) {  //用康托展开判重
    long result = 0;
    for(int i = 0; i < n; i++) {
        int counted = 0;
        for(int j = i+1; j < n; j++) {
            if(str[i] > str[j])  //当前未出现的元素排在第几个
                ++counted;
        }
        result += counted * factory[n-i-1];
    }
    if(!visited[result]) {       //没有被访问过
        visited[result] = 1;
        return 1;
    }
    else
        return 0;
}
int bfs() {
    node head;
    memcpy(head.state, start, sizeof(head.state));    //复制起点的状态
    head.dis = 0;
    queue<node> q;               //队列中的内容是记录状态
    Cantor(head.state, 9);       //用康托展开判重，目的是对起点的 visited[]赋初值
    q.push(head);                //第一个进队列的是起点状态

    while(!q.empty()) {          //处理队列
        head = q.front();
        q.pop();                 //可在此处打印 head.state，看弹出队列的情况
        int z;
        for(z = 0; z < 9; z++)   //找这个状态中元素 0 的位置
            if(head.state[z] == 0)//找到了
```

① 本题中的队列比较简单，如果不用 STL，也可以用简单的方法模拟队列，请搜索网上的代码。

```
                break;
            //转化为二维,左上角是原点(0,0)
        int x = z%3;                        //横坐标
        int y = z/3;                        //纵坐标
        for(int i = 0; i<4; i++){           //上、下、左、右最多可能有 4 个新状态
            int newx = x+dir[i][0];         //元素 0 转移后的新坐标
            int newy = y+dir[i][1];
            int nz = newx + 3*newy;         //转化为一维
            if(newx>=0 && newx<3 && newy>=0 && newy<3) {    //未越界
                node newnode;
                memcpy(&newnode,&head,sizeof(struct node));  //复制这新的状态
                swap(newnode.state[z], newnode.state[nz]);   //把 0 移动到新的位置
                newnode.dis ++;
                if(memcmp(newnode.state,goal,sizeof(goal)) == 0)
                                        //与目标状态对比
                    return newnode.dis;         //到达目标状态,返回距离,结束
                if(Cantor(newnode.state, 9))    //用康托展开判重
                    q.push(newnode);            //把新的状态放进队列
            }
        }
    }
    return -1;                              //没找到
}
int main(){
    for(int i = 0; i<9; i++)  cin>>start[i];    //初始状态
    for(int i = 0; i<9; i++)  cin>>goal[i];     //目标状态
    int num = bfs();
    if(num != -1)   cout<<num<<endl;
    else            cout<<"Impossible"<<endl;
    return 0;
}
```

上述代码的细节很多,请读者仔细体会,要求能独立写出来。

15 数码问题。八数码问题只有 9!种状态,对于更大的问题,例如 4×4 棋盘的 15 数码问题,有 $16!\approx2\times10^{13}$ 种状态,如果仍然用数组存储状态,远远不够,此时需要更好的算法[①]。

【习题】

poj 1077 “Eight”,八数码问题。另外,在学过下一节的 A * 算法后可重新做这道题。

4.3.3 BFS 与 A * 算法

1. 用 BFS 求最短路径

最短路径是图论的一个基本问题,有很多复杂的算法。不过,在特殊的地图中,BFS 也是很好的最短路径算法。下面仍然以 hdu 1312"Red and Black"的方格图为例,任务是求两点之间的最短路径。

① 八数码的多种解法,例如双向广搜、A * 、IDA * 等,请参考“https://www.cnblogs.com/zufezzt/p/5659276.html”(永久网址: perma.cc/YV2V-GT6C)。

在图 4.4 中,黑点“•”表示可以走的路,“#”表示不能走。求起点“@”到所有黑点“•”的最短距离。

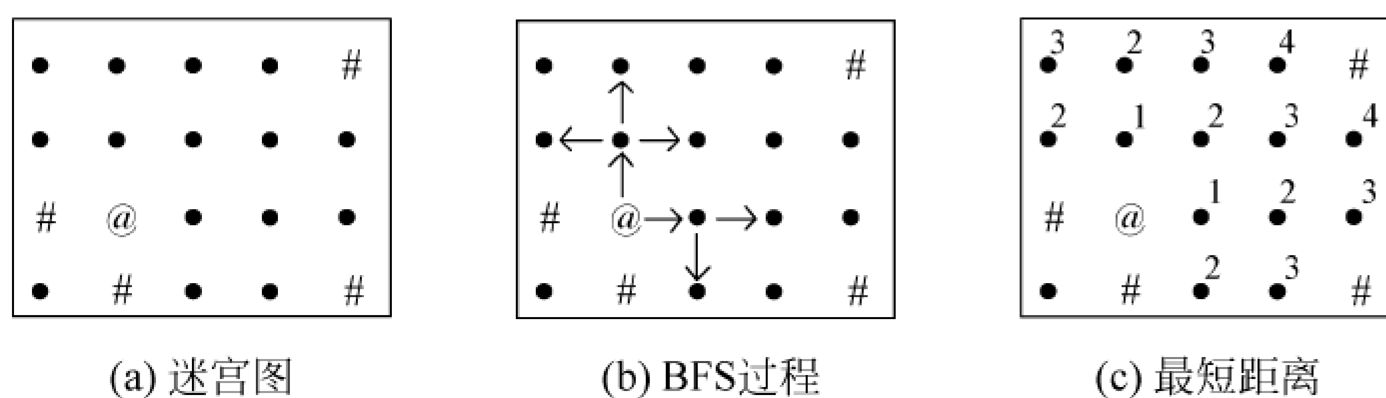


图 4.4 用 BFS 找最短路径

方法很简单,从“@”出发,用 BFS 搜索所有点,记录到达每个点时经过的步数,即可得到从“@”到所有黑点的最短距离,图 4.4(c)标出了结果。

在这个例子中,BFS 搜最短路径的计算复杂度是 $O(V+E)$,非常好。

这个例子很特殊,图是方格形的,相邻两点之间的距离相同。也就是说,绕路肯定更远;BFS 先扩展到的路径,距离肯定是最短的。

如果相邻点的距离不同,绕路可能更近,BFS 就不适用了。关于最短路径的通用算法,请阅读本书的 10.9 节的内容。

下面的 A * 算法是 BFS 的优化。

2. A * 算法与最短路径

BFS 是一种“盲目的”搜索技术,它在搜索的过程中并不理会目标在哪里,只顾自己乱走,当然最后总会到达终点。

稍微改变 hdu 1312 的方格图,见图 4.5(a),现在的任务是求起点“@”到终点“t”的最短路径。

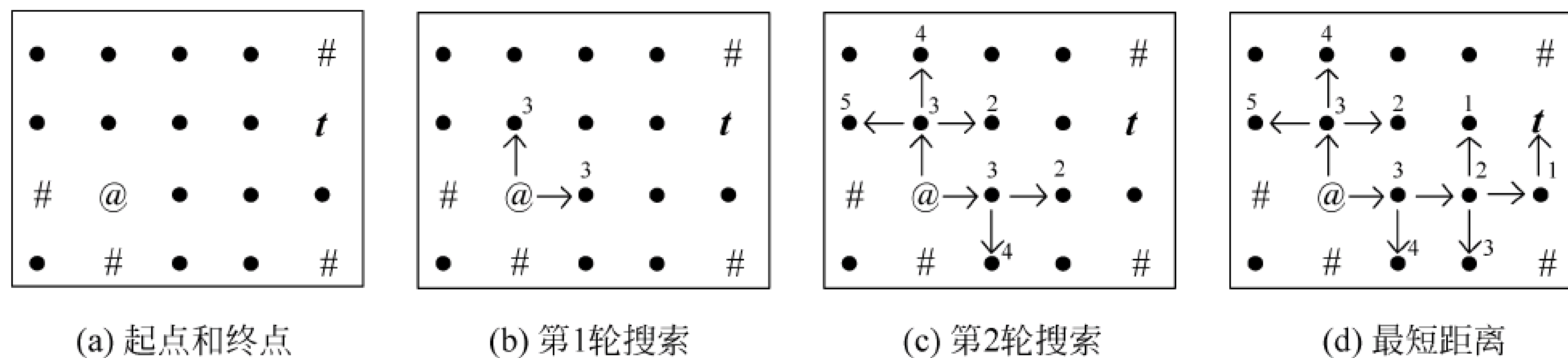


图 4.5 启发式搜索

如果仍然用 BFS 求解,程序会搜索所有的点,直到遇到 t 点。不过,如果让一个人走这个图,他会一眼看出向右上方走可以更快地找到到达 t 的最短路径。人有“智能”,那么能否把这种智能教给程序呢? 这就是“启发式”搜索算法。启发式搜索算法有很多种,A * 算法是其中比较简单的一种。

简单地说,A * 算法是“BFS+贪心”[①]。有关贪心法的解释,请阅读本书的 6.1 节。

在图 4.5(a)中,程序如何知道向右上方走能更快地到达 t?这里引入曼哈顿距离的概念。曼哈顿距离是指两个点在标准坐标系上的实际距离,在图 4.5 中就是@的坐标和 t 的

① 这个网页用动画演示了 BFS、A * 、Dijkstra 算法的原理,并给出了比较详细的伪代码描述,非常值得一看,网址为 https://www.redblobgames.com/pathfinding/a-star/introduction.html(永久网址:https://perma.cc/N2DB-5LDY)。

坐标在横向和纵向的距离之和，它也被形象地称为“出租车距离”。

图 4.5(b)是从起点开始的第 1 轮 BFS 搜索，邻居点上标注的数字 3 是这个点到终点 t 的曼哈顿距离。图 4.5(c)是第 2 轮搜索，标注 2 的点是离终点更近的点，从这些点继续搜索；标注 4 和 5 的点距离终点远，先暂时停止搜索。经过多轮搜索，最后到达了终点 t，如图 4.5(d)所示。

在这个过程中，图中很多“不好的”点并不需要搜索到，从而优化了搜索过程。

上面的图例比较简单，如果起点和终点之间有很多障碍，搜索范围也会沿着障碍兜圈子，之后才能到达终点，不过，仍然有很多点不需要搜索。以下面的图 4.6 为例，A 是起点，B 是终点，黑色方块是障碍，浅色阴影方块是用曼哈顿距离进行启发式搜索所经过的部分，其他无色方块是不需要搜索的。搜索结束后，得到一条最短路径，见图中的虚线。

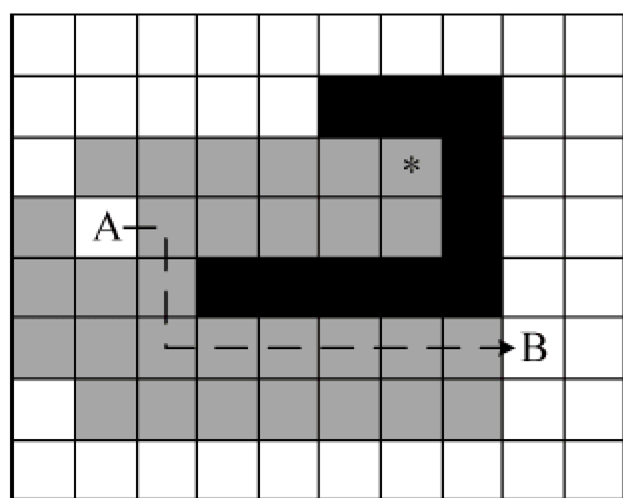


图 4.6　A * 搜索求最短路径


视频讲解

这个方法就是 A * 算法，下面给出它的一般性描述。

在搜索过程中，用一个评估函数对当前情况进行评估，得到最好的状态，从这个状态继续搜索，直到目标。设 x 是当前所在的状态，$f(x)$是对 x 的评估函数，有：

$$f(x)=g(x)+h(x)$$

$g(x)$表示从初始状态到 x 的实际代价，它不体现 x 和终点的关系。

$h(x)$表示 x 到终点的最优路径的评估，它就是“启发式”信息，把 $h(x)$称为启发函数。很显然，$h(x)$决定了 A * 算法的优劣。

特别需要注意的是，$h(x)$不能漏掉最优解。

在上面的例子中，曼哈顿距离就是启发函数 $h(x)$。曼哈顿距离是一种简单而且常用的启发函数。

在上面这个例子中，可以看出 A * 算法包含了 BFS 和贪心算法。

(1) 如果 $h(x)=0$，有 $f(x)=g(x)$，就是普通的 BFS 算法，会访问大量的方块。

(2) 如果 $g(x)=0$，有 $f(x)=h(x)$，就是贪心算法，此时图中标注“ * ”的方块也会被访问到。贪心法的缺点是可能陷在局部最优中，例如陷在“ * ”的方块中，被堵在障碍后面，无法到达终点。

3. A * 算法与八数码问题

八数码问题也可以用 A * 算法进行优化。通常考虑 3 种估价函数：

(1) 以不在目标位置的数码的个数作为估价函数。

(2) 以不在目标位置的数码与目标位置的曼哈顿距离作为估价函数。

(3) 以逆序数[①]作为估价函数。

第(2)种比第(1)种好，可作为八数码问题的估价函数。

4.3.4　双向广搜

双向广搜是 BFS 的增强版。

① 逆序数可以用来判断八数码是否有解。

前面提到,可以把 BFS 想象成在一个平静的池塘丢一颗石头,激起的波浪一层层扩散到整个空间,直到到达目标,就得到了从起点到目标点的最优路径。那么,如果同时在起点和目标点向对方做 BFS,两个石头激起的波浪向对方扩散,将在中间的某个位置遇到,此时即得到了最优路径。在绝大多数情况下,双向广搜比只做一次 BFS 搜索的空间要少很多,从而更有效率。

从上面的描述可知,双向广搜的应用场合是知道起点和终点,并且正向和逆向都能进行搜索。

下面是一个典型的双向广搜问题。

> **hdu 1401 "Solitaire"**
>
> 有一个 8×8 的棋盘,上面有 4 颗棋子,棋子可以上下左右移动。给定一个初始状态和一个目标状态,问能否在 8 步之内到达。

题目确定了起点和终点,十分适合双向 BFS。要求在 8 步之内到达,可以从起点和终点分别开始,各自广搜 4 步,如果出现交点则说明可达。读者可以练习此题,虽然程序比较烦琐,有很多细节需要处理,但是难度不高。

4.3.2 节讲解的八数码问题也非常适合使用双向广搜技术进行优化。

【习题】

hdu 1401 "Solitaire";

hdu 3567 "Eight II",用双向广搜解决八数码问题。

4.4 DFS

4.4.1 DFS 和递归

hdu 1312 题有另外一种解决方案,即 4.3.1 节中提到的"一只老鼠走迷宫"。设 num 是到达的砖块数量,算法过程描述如下:

(1) 在初始位置令 num=1,标记这个位置已经走过。

(2) 左、上、右、下 4 个方向,按顺时针顺序选一个能走的方向,走一步。

(3) 在新的位置 num++,标记这个位置已经走过。

(4) 继续前进,如果无路可走,回退到上一步,换个方向再走。

(5) 继续以上过程,直到结束。

在以上过程中,能够访问到所有合法的砖块,并且每个砖块只访问一次,不会重复访问(回退不算重复),如图 4.7 所示。

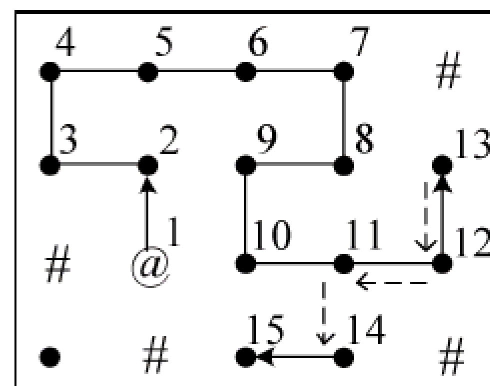


图 4.7 DFS 过程

hdu 1312 的路线如下:从 1 到 13,能一直走下去。在 13 这个位置,到底了不能再走,按顺序回退到 12、11;在 11 这个位置,换个方向又能走到 14、15。到达 15 后,发现不能再走下去,那么按顺

序倒退，即 14→11→10→9→8→7→6→5→4→3→2→1，在这个过程中发现全部都没有新路，最后退回到起点，结束。

为加深对递归的理解，这里再次给出递归返回的完整顺序，即 13→12→15→14→11→10→9→8→7→6→5→4→3→2→1。

在这个过程中，最重要的特点是在一个位置只要有路，就一直走到最深处，直到无路可走，再退回一步，看在上一步的位置能不能换个方向继续往下走。这样就遍历了所有可能走到的位置。

这个思路就是深度搜索。从初始状态出发，下一步可能有多种状态；选其中一个状态深入，到达新的状态；直到无法继续深入，回退到前一步，转移到其他状态，然后再深入下去。最后，遍历完所有可以到达的状态，并得到最终的解。

上述过程用 DFS 实现是最简单的，代码比 BFS 短很多。

下面是代码。读者可以在 DFS()函数中打印走过的位置以及回退的情况。从打印的信息可以看出，在到达 15 后，程序确实是逐步回退到起点的。

```
//用 DFS()替换 4.3.1 节程序中的 BFS(),并在 main()中的相同位置调用它
void DFS(int dx, int dy){
    room[dx][dy] = '#';                  //标记这个位置,表示已经走过
      //cout <<"walk:"<< dx << dy << endl; //在此处打印走过的位置,验证是否符合
    num++;
    for(int i = 0; i < 4; i++) {         //左、上、右、下 4 个方向顺时针深搜
        int newx = dx + dir[i][0];
        int newy = dy + dir[i][1];
        if(CHECK(newx, newy) && room[newx][newy] == '.'){
            DFS(newx, newy);
            //cout <<"     back:"<< dx << dy << endl;
            //在此处打印回退的点的坐标,观察深搜到底后回退的情况
            //例如到达最后的 15 这个位置后会一直退到起点
            //即打印出 14-11-10-9-8-7-6-5-4-3-2-1.这也是递归程序返回的过程
        }
    }
}
```

4.4.2 回溯与剪枝

前面提到的 DFS 搜索，基本的操作是将所有子结点全部扩展出来，再选取最新的一个结点进行扩展。

不过，在很多情况下，用递归列举出所有的路径可能会因为数量太大而超时。由于很多子结点是不符合条件的，可以在递归的时候“看到不对头就撤退”，中途停止扩展并返回。这个思路就是回溯，在回溯中用于减少子结点扩展的函数是剪枝函数。

大部分 DFS 搜索题目都需要用到回溯的思路，其难度主要在于扩展子结点的时候如何构造停止递归并返回的条件。这需要通过大量地练习有关题目才能熟练应用。

八皇后问题是经典的回溯与剪枝的应用。

八皇后问题。在棋盘上放置 8 个皇后，使得它们不同行、不同列、不同对角线。N 皇后问题是八皇后问题的扩展。

如果用暴力方法，先排列出所有的棋局，然后一一判断、去除非法的棋局，请读者自己思考复杂度有多大。

下面以四皇后问题为例描述解题过程。在图 4.8 中，从第 1 行开始放皇后：第 1 行从左到右有 4 种方案，产生 4 个子结点；第 2 行，排除同列和斜线，扩展新的子结点，注意不用排除同行，因为第 2 行和第 1 行已经不同行；继续扩展第 3 行和第 4 行，结束。

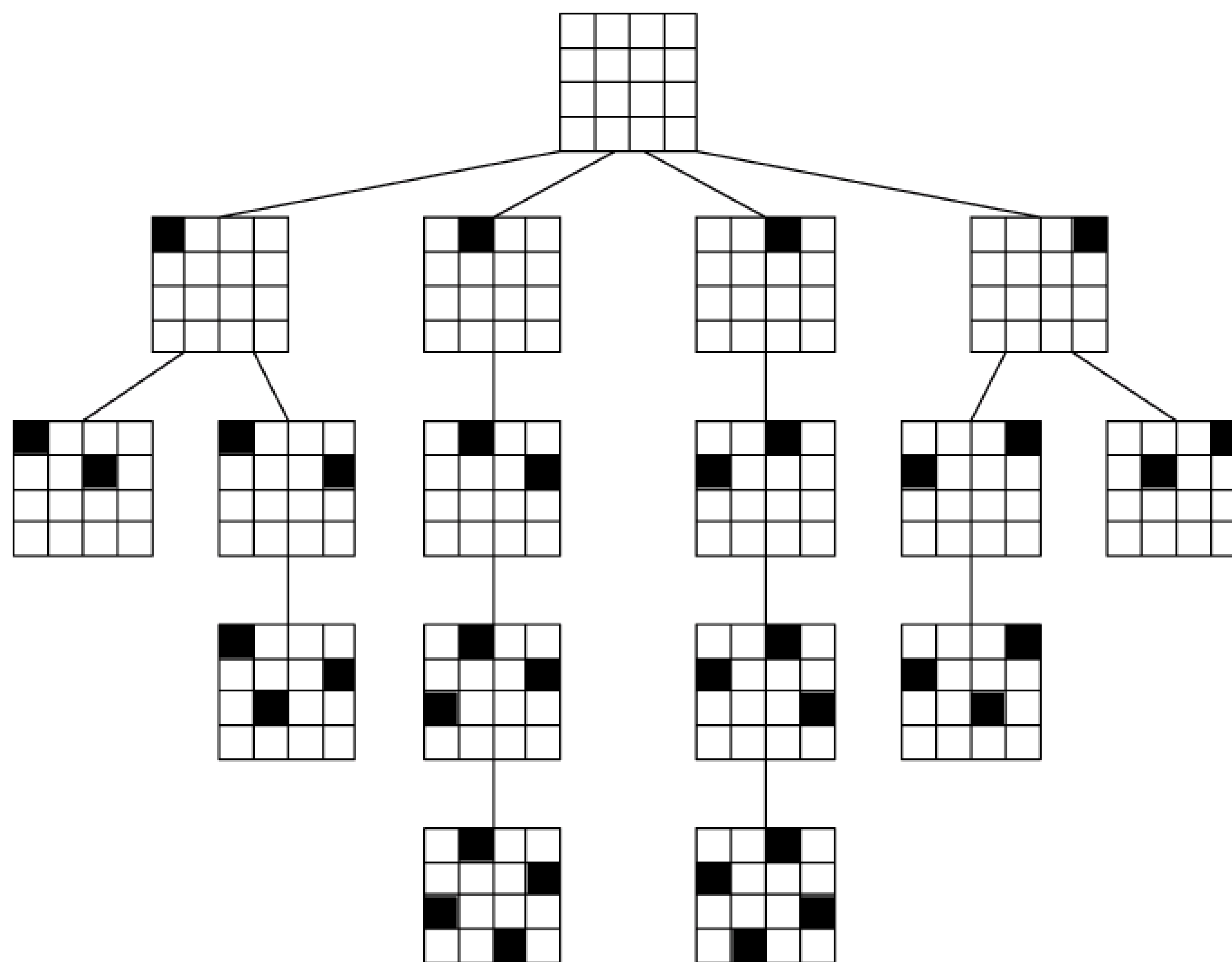


图 4.8　四皇后问题的搜索树

该图用 BFS 和 DFS 都能实现。前文说过，DFS 的代码比 BFS 简洁很多。下面用 DFS 来解决。

关键问题：在扩展结点时如何去掉不符合条件的子结点？

设左上角是原点(0,0)，已经放好的皇后的坐标是(i,j)，不同行、不同列、不同斜线的新皇后的坐标是(c,r)，它们的关系如下：

(1) 横向，不同行：$i \neq r$。

(2) 纵向，不同列：$j \neq c$。

(3) 斜对角：从(i,j)向斜对角走 a 步，那么新坐标(r,c)有 4 种情况，即左上$(i-a,j-a)$、右上$(i+a,j-a)$、左下$(i-a,j+a)$、右下$(i+a,j+a)$，综合起来就是$|i-r|=|j-c|$。新皇后的位置不能放在斜线上，需满足$|i-r| \neq |j-c|$。

下面是 hdu 2553 的代码，求解 N 皇后问题，$N \leqslant 10$。

hdu 2553 "*N* 皇后问题"

```
#include <bits/stdc++.h>
using namespace std;
int n, tot = 0;
int col[12] = {0};
bool check(int c, int r) {                  //检查是否和已经放好的皇后冲突
    for(int i = 0; i < r; i++)
        if(col[i] == c || (abs(col[i] - c) == abs(i - r)))      //取绝对值
            return false;
```

```
        return true;
    }
    void  DFS(int r) {                        //一行一行地放皇后,这一次是第 r 行
        if(r == n) {                          //所有皇后都放好了,递归返回
            tot++;                            //统计合法的棋局个数
            return;
        }
        for(int c = 0; c < n; c++)            //在每一列放皇后
            if(check(c, r)){                  //检查是否合法
                col[r] = c;                   //在第 r 行的 c 列放皇后
                DFS(r + 1);                   //继续放下一行皇后
            }
    }
    int main() {
        int ans[12] = {0};
        for(n = 0; n <= 10; n++){             //算出所有 N 皇后的答案.先打表,不然会超时
            memset(col,0,sizeof(col));        //清空,准备计算下一个 N 皇后问题
            tot = 0;
            DFS(0);
            ans[n] = tot;                     //打表
        }
        while(cin >> n) {
            if(n == 0)
                return 0;
            cout << ans[n] << endl;
        }
        return 0;
    }
```

N 皇后问题的 DFS 回溯程序非常简单,关键有两处,一是如何递归,二是如何剪枝和回溯。在上述程序中有很多细节,例如:

(1) 打表。在 main()中提前算出了从 1 到 10 的所有 N 皇后问题的答案,并存储在数组中,等读取输入后立刻输出。如果不打表,而是等输入 N 后再单独计算输出,会超时。

(2) 递归搜索 DFS()。递归程序十分简洁,把第 1 个皇后按行放到棋盘上,然后递归放置其他的皇后,直到放完。

(3) 回溯判断 check()。判断新放置的皇后和已经放好的皇后在横向、纵向、斜对角方向是否冲突。其中,横向并不需要判断,因为在递归的时候已经是按不同的行放置的。

(4) 模块化编程。例如 check()的内容很少,其实可以直接写在 DFS()内部,不用单独写成一个函数。但是单独写成函数,把功能模块化,好处很多,例如逻辑清晰、容易查错等。建议在写程序的时候尽量把能分开的功能单独写成函数,这样可以大大减少编码和调试的时间。

(5) 复杂度。在上述程序中,DFS()一行行地放皇后,复杂度为 $O(N!)$;check()检查冲突,复杂度为 $O(N)$;总复杂度为 $O(N\times N!)$。当 $N=10$ 时,已经到千万数量级。读者可以自己在程序中统计运行次数。经本书作者验证,$N=11$ 时计算了 900 万次,$N=12$ 时计算了 5 千万次。因此,对于 $N>11$ 的 N 皇后问题,需要用新的方法[①]。

① 用数据结构舞蹈链(Dancing Links)或者位运算可以较快地解决 $N=15$ 的 N 皇后问题。对于更大的 N,例如当 $N=27$ 时,有 2.34×10^{17} 个解。N 皇后问题是一个 NP 完全问题,不存在多项式时间的算法。

【习题】

poj 2531 “Network Saboteur”；
poj 1416 “Shredding Company”；
poj 2676 “Sudoku”；
poj 1129 “Channel Allocation”；
hdu 1175 “连连看”；
hdu 5113 “Black And White”。

4.4.3　迭代加深搜索

有这样一些题目，它们的搜索树很特别：不仅很深，而且很宽；深度可能到无穷，宽度也可能极广。如果直接用 DFS，会陷入递归无法返回；如果直接用 BFS，队列空间会爆炸。

此时可以采用一种结合了 DFS 和 BFS 思想的搜索方法，即迭代加深搜索（Iterative Deepening DFS，IDDFS）。具体的操作方法如下：

（1）先设定搜索深度为 1，用 DFS 搜索到第 1 层即停止。也就是说，用 DFS 搜索一个深度为 1 的搜索树。

（2）如果没有找到答案，再设定深度为 2，用 DFS 搜索前两层即停止。也就是说，用 DFS 搜索一个深度为 2 的搜索树。

（3）继续设定深度为 3、4……逐步扩大 DFS 的搜索深度，直到找到答案。

这个迭代过程，在每一层的广度上采用了 BFS 搜索的思想，在具体编程实现上则是 DFS 的。

一个经典的例子是“埃及分数”。

埃及分数[①]

在古埃及，人们使用单位分数的和（形如 $1/a$ 的，a 是自然数）表示一切有理数。例如 $2/3=1/2+1/6$，但不允许 $2/3=1/3+1/3$，因为加数中有相同的。对于一个分数 a/b，表示方法有很多种，但是哪种最好呢？首先，加数少的比加数多的好，其次，加数个数相同的，最小的分数越大越好。例如：

$$19/45=1/3+1/12+1/180$$
$$19/45=1/3+1/15+1/45$$
$$19/45=1/3+1/18+1/30$$
$$19/45=1/4+1/6+1/180$$
$$19/45=1/5+1/6+1/18$$

最好的是最后一种，因为 1/18 比 1/180、1/45、1/30 都大。给出 a、b（$0<a<b<1000$），编程计算最好的表达方式。

① http://codevs.cn/problem/1288/。

这一题显然是搜索，可以按图 4.9 建立搜索树。每一层的元素是分子为 1、分母递增的分数；从上往下的一个分支，就是一个这个分支上所有的分数相加的组合；找到合适的组合就退出。解答树的规模很大，深度可能无限，每一层的宽度也可能无限。

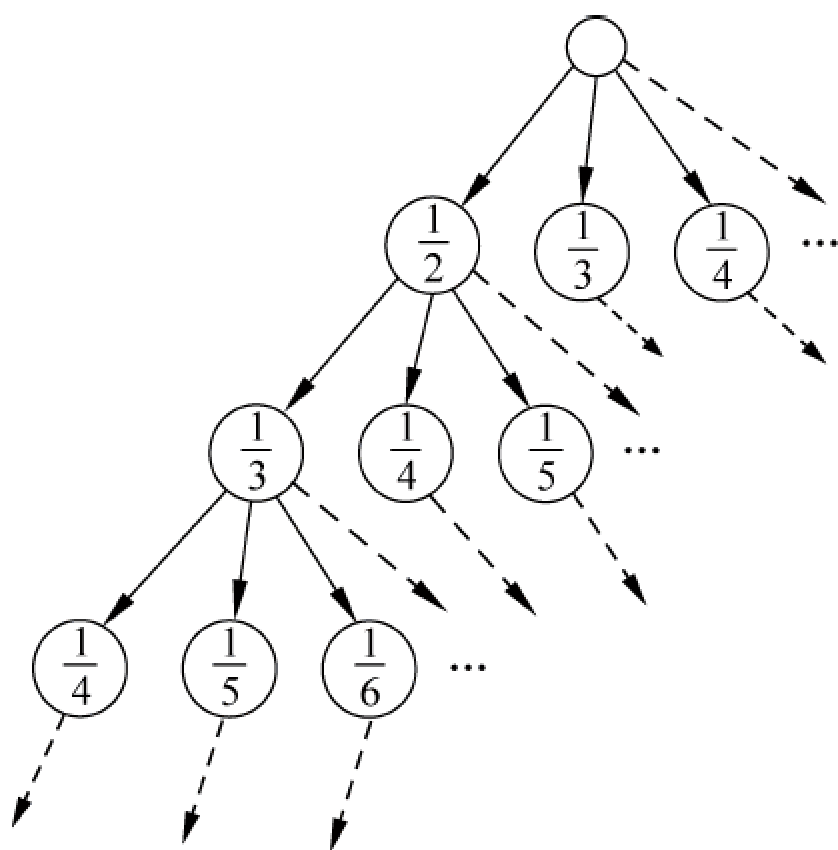


图 4.9　深度和宽度极大的搜索树

在这种情况下适合使用迭代加深搜索。其过程如下：

(1) DFS 到第 1 层，只包括一个分数，如果满足要求就退出。

(2) DFS 前两层，是两个分数的和，例如 1/2+1/3、1/2+1/4、1/2+1/5、…、1/3+1/4、…，找到合适的答案就退出。

(3) DFS 前 3 层……

按上述步骤能搜索到所有可能的组合，并且规避了直接使用 DFS 或 BFS 的弊端。

4.4.4　IDA *

IDA * 是对迭代加深搜索 IDDFS 的优化，可以把 IDA * 看成 A * 算法思想在迭代加深搜索中的应用。

IDDFS 仍然是一种"盲目"的搜索方法，只是把搜索范围约束到了可行的空间内。如果在进行 IDDFS 的时候能预测出当前 DFS 的状态，不再继续深入下去，那么就可以直接返回，不再继续，从而提高了效率。

这个预测就是在 IDDFS 中增加一个估价函数。在某个状态，经过函数计算，发现后续搜索无解，就返回。简单地说，就是在 IDDFS 的过程中利用估价函数进行剪枝操作。

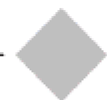
下面这个例题说明了 IDDFS 和估价函数之间的关系。

poj 3134 "Power Calculus"

给定数 x 和 n，求 x^n，只能用乘法和除法，算过的结果可以被利用。问最少算多少次就够了。其中 $n \leqslant 1000$。

这一题等价于从数字 1 开始，用加减法，最少算多少次能得到 n。

搜索的范围是每一步搜索，用前一步得出的值和之前产生的所有值进行加、减运算得到



新的值，判断这个值是否等于 n。

这一题的麻烦在于，每一步搜索，新值的数量增长极快。如果直接用 DFS，深度可能有 1000，可能会溢出；如果用 BFS，也可能超出队列范围。

这一题用 IDDFS 非常合适，再用估价函数进行剪枝，可以高效地完成计算。

(1) IDDFS：指定递归深度，每一次做 DFS 时不超过这个深度。

(2) 估价函数：如果当前的值用最快的方式(连续乘 2，倍增)都不能到达 n，停止用这个值继续 DFS。

poj 3134 的代码

```
#include <iostream>
using namespace std;
int val[1010];                              //保存一个搜索路径上每一步的计算结果
int pos, n;
bool ida(int now, int depth){
    if(now > depth)   return false;         //IDDFS:大于当前设定的 DFS 深度,退出
    if(val[pos] << (depth - now) < n)
        return false;                       //估价函数:用最快的倍增都不能到达 n,退出
    if(val[pos] == n)   return true;        //当前结果等于 n,搜索结束
    pos ++ ;
    for(int i = 0 ; i < pos ; i ++) {
        val[pos] = val[pos - 1] + val[i];       //上一个数与前面所有的数相加
        if(ida(now + 1, depth))   return true;
        val[pos] = abs(val[pos - 1] - val[i]);  //上一个数与前面所有的数相减
        if(ida(now + 1, depth))   return true;
    }
    pos -- ;
    return false;
}
int main(){
    int t;
    while(cin >> n && n){
        int depth;
        for(depth = 0 ; ; depth ++){            //每次只 DFS 到深度 depth
            val[pos = 0] = 1;                   //初始值是 1
            if(ida(0, depth)) break;            //每次都从 0 层开始 DFS 到第 depth 层
        }
        cout << depth << endl;
    }
    return 0;
}
```

【习题】

hdu 1560 “DNA sequence”，经典 IDA * 题目；

hdu 1667 “The Rotation Game”，经典 IDA * 题目。

4.5 小　　结

DFS 和 BFS 是算法设计中的基本技术，是基础的基础。

这两种算法都能遍历搜索树的所有结点，区别在于如何扩展下一个结点。DFS 扩展子结点的子结点，搜索路径越来越深，适合采用栈这种数据结构，并用递归算法来实现；BFS 扩展子结点的兄弟结点，搜索路径越来越宽，适合用队列来实现。

1. 复杂度

DFS 和 BFS 对所有的点和边做了一次遍历，即对每个结点均做一次且仅做一次访问。设点的数量是 V，连接点的边总数是 E，那么总复杂度是 $O(V+E)$，看起来复杂度并不高。但是，有些问题的 V 和 E 本身就是指数级的，例如八数码问题的状态，是 $O(n!)$的。因此，在搜索时需要用到剪枝、回溯、双向广搜、迭代加深、A *、IDA * 等方法，尽量减少搜索的范围，使访问的总次数远远小于 $O(V+E)$。

2. 应用场合

DFS 一般用递归实现，代码比 BFS 更短。如果题目能用 DFS 解决，可以优先使用它。

当然，一些问题更适合用 DFS，另一些问题更适合用 BFS。在一般情况下，BFS 是求解最优解的较好方法，例如像迷宫这样的求最短路径问题应该用 BFS，具体内容见第 10 章中的“最短路径”；而 DFS 多用于求可行解。在第 10 章中还有大量应用 BFS 和 DFS 的例子。

第 5 章　高级数据结构

✍ 并查集
✍ 二叉树
✍ Treap 树
✍ Splay 树
✍ 线段树
✍ 树状数组

竞赛题是对输入的数据进行运算，然后输出结果。因此，编写程序的一个基本问题就是数据处理，包括如何存储输入的数据、如何组织程序中的中间数据等。这个技术就是数据结构。学习数据结构，建立计算思维的基础，是成为合格程序员的基本功。本章讲解一些常用的高级数据结构。

数据结构的作用是分析数据、组织数据、存储数据。基本的数据类型有字符和数字，这些数据需要存储在空间中，然后程序按规则读取和处理它们。

数据结构和算法不同，它并不直接解决问题，但是数据结构是算法不可分割的一部分。首先，数据结构把杂乱无章的数据有序地组织起来，逻辑清晰，易于编程处理；其次，数据结构便于算法高效地访问和处理数据，大大减少空间和时间复杂度。

(1) 存储的空间效率。例如一个围棋程序，需要存储棋盘和棋盘上棋子的位置。棋盘可以简单地用一个 19×19 的二维数组(矩阵)表示。每个棋子是一个坐标，例如 $W[5][6]$ 表示位于第 5 行第 6 列的白棋。这种二维数组是数据结构“图”的一种描述，图是描述点和点之间连接关系的数据结构。棋盘只是一种简单的图，更复杂的图例如地图。地图上有两种元素，即点、点之间直连的道路。地图比棋盘复杂，棋盘的每个点只有上、下、左、右 4 个相邻的点，而地图上的一个点可能有很多相邻的点。那么如何存储一个地图？可以简单地用一个二维数组，例如有 n 个点，用一个 $n\times n$ 的二维矩阵表示地图，矩阵上的交叉点 (i,j) 表示第 i 点和第 j 点的连接关系，例如 1 表示相邻，0 表示不相邻。二维矩阵这种数据结构虽然简单、访问速度快，但是用它来存储地图非常浪费空间，因为这是一个稀疏矩阵，其中的交叉点绝大多数等于 0，这些等于 0 的交叉点并不需要存储。一个有 10 万个点的地图，存储它的二维矩阵大小是 $100\ 000\times100\ 000=10\text{GB}$。所以，在程序中使用二维矩阵来存储地图是不行的，例如手机上的导航软件，常常有几十万个地点，手机存储卡根本放不下。因此，大地图的存储需要用到更有效率的数据结构，这就是邻接表。

(2) 访问的效率。例如输入一大串个数为 n 的无序数字，如果直接存储到一个一维数组里面，那么要查找到某个数据，只能一个个试，需要的时间是 $O(n)$。如果先按大小排序然后再查询，处理起来就很有效率。在 n 个有序的数中找某个数，用折半查找的方法，可以在

$O(\log_2 n)$的时间里找到。

用数据结构存储和处理数据，可以使程序的逻辑更加清晰。

数据结构有以下 3 个要素[①]。

(1) 数据的逻辑结构：线性结构(数组、栈、队列、链表)、非线性结构、集合、图等。

(2) 数据的存储结构：顺序存储(数组)、链式存储、索引存储、散列存储等。

(3) 数据的运算：初始化、判空、统计、查找、遍历、插入、删除、更新等。

常见的数据结构有数组、链表、栈、队列、树、二叉树、集合、哈希、堆与优先队列、并查集、图、线段树、树状数组等。

在第 3 章中已经介绍了基本的数据结构[②]——栈、队列、链表，本章继续讲解一些常用的高级数据结构，包括并查集、二叉树、线段树、树状数组。

5.1 并 查 集

并查集(Disjoint Set)是一种非常精巧而且实用的数据结构，它主要用于处理一些不相交集合的合并问题。经典的例子有连通子图、最小生成树 Kruskal 算法[③]和最近公共祖先(Lowest Common Ancestors，LCA)等。

通常用“帮派”的例子来说明并查集的应用背景。在一个城市中有 n 个人，他们分成不同的帮派；给出一些人的关系，例如 1 号、2 号是朋友，1 号、3 号也是朋友，那么他们都属于一个帮派；在分析完所有的朋友关系之后，问有多少帮派，每人属于哪个帮派。给出的 n 可能是 10^6 的。

读者可以先思考暴力的方法以及复杂度。如果用并查集实现，不仅代码很简单，而且复杂度可以达到 $O(\log_2 n)$。

并查集：将编号分别为 $1\sim n$ 的 n 个对象划分为不相交集合，在每个集合中，选择其中某个元素代表所在集合。在这个集合中，并查集的操作有初始化、合并、查找。

下面先给出并查集操作的简单实现。在这个基础上，后文再进行优化。



1. 并查集操作的简单实现

(1) 初始化。定义数组 int s[]是以结点 i 为元素的并查集，在开始的时候还没有处理点与点之间的朋友关系，所以每个点属于独立的集，并且以元素 i 的值表示它的集 $s[i]$，例如元素 1 的集 $s[1]=1$。

图 5.1 所示为图解，左边给出了元素与集合的值，右边画出了逻辑关系。为了便于讲解，左边区分了结点 i 和集 s(把集的编号加上了下画线)；右边用圆圈表示集，方块表示元素。

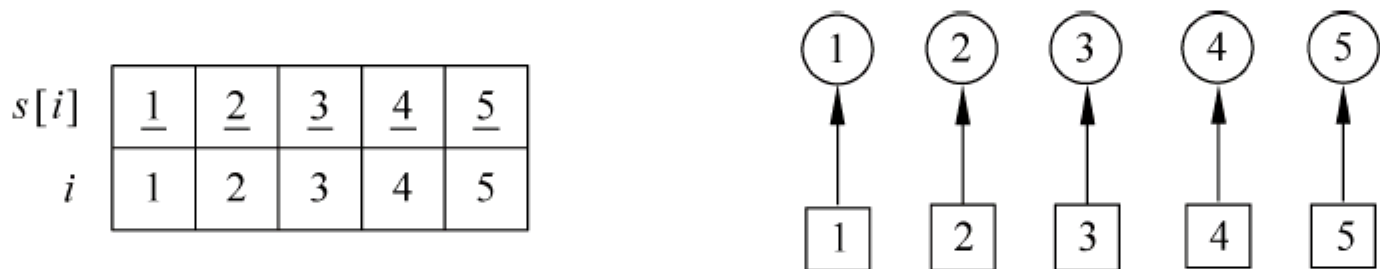


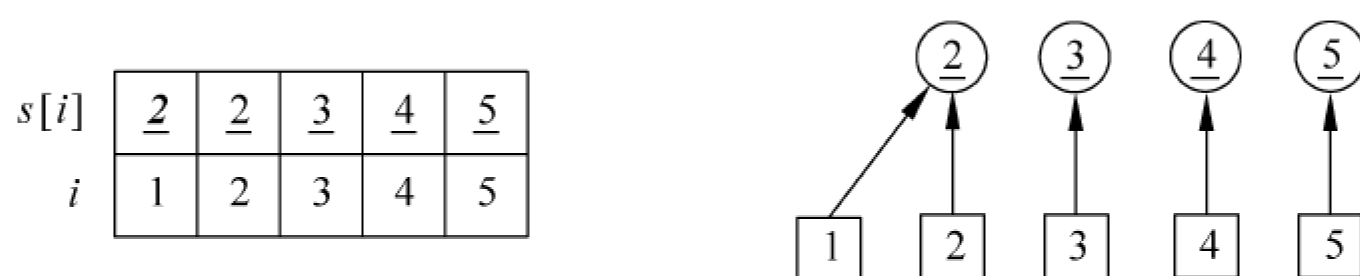
图 5.1 并查集的初始化

① 《数据结构与算法分析新视角》，作者周幸妮等，电子工业出版社。

② 《数据结构(STL 框架)》，作者王晓东，清华大学出版社。

③ 参考本书的“10.10.2 kruskal 算法”。

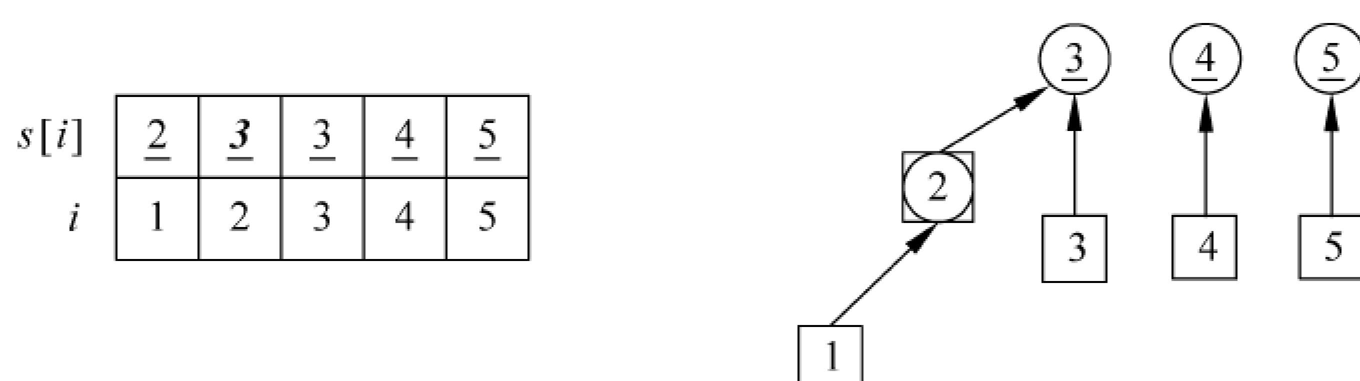
(2) 合并,例如加入第 1 个朋友关系(1,2),如图 5.2 所示。在并查集 s 中,把结点 1 合并到结点 2,也就是把结点 1 的集1 改成结点 2 的集2。



$s[i]$	2	2	3	4	5
i	1	2	3	4	5

图 5.2 合并(1,2)

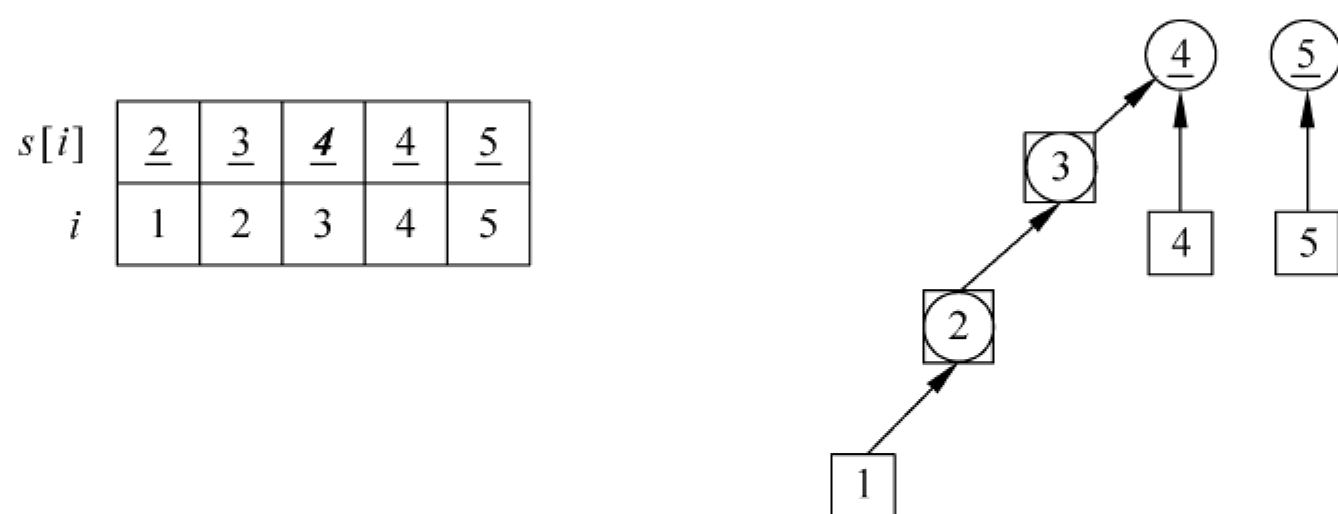
(3) 合并,加入第 2 个朋友关系(1,3),如图 5.3 所示。查找结点 1 的集是2,再递归查找元素 2 的集是2,然后把元素 2 的集2 合并到结点 3 的集3。此时,结点 1、2、3 属于一个集。在右图中,为了简化图示,把元素 2 和集2 画在了一起。



$s[i]$	2	3	3	4	5
i	1	2	3	4	5

图 5.3 合并(1,3)

(4) 合并,加入第 3 个朋友关系(2,4),如图 5.4 所示。结果如下,请读者自己分析。



$s[i]$	2	3	4	4	5
i	1	2	3	4	5

图 5.4 合并(2,4)

(5) 查找。在上面步骤中已经有查找操作。查找元素的集是一个递归的过程,直到元素的值和它的集相等就找到了根结点的集。从上面的图中可以看到,这棵搜索树的高度可能很大,复杂度是 $O(n)$ 的,变成了一个链表,出现了树的"退化"现象。

(6) 统计有多少个集。如果 $s[i]=i$,这是一个根结点,是它所在的集的代表;统计根结点的数量,就是集的数量。

下面以 hdu 1213 为例实现上述操作。

---

hdu 1213 "How Many Tables"

有 n 个人一起吃饭,有些人互相认识。认识的人想坐在一起,不想跟陌生人坐。例如 A 认识 B,B 认识 C,那么 A、B、C 会坐在一张桌子上。

给出认识的人,问需要多少张桌子。

一张桌子是一个集,合并朋友关系,然后统计集的数量即可。下面的代码是并查集操作

的具体实现。

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 1050;
int s[maxn];
void init_set(){                          //初始化
   for(int i = 1; i <= maxn; i++)
        s[i] = i;
}
int find_set(int x){                      //查找
    return x == s[x]? x:find_set(s[x]);
}
void union_set(int x, int y){             //合并
    x = find_set(x);
    y = find_set(y);
    if(x != y) s[x] = s[y];
}
int main(){
    int t, n, m, x, y;
    cin >> t;
    while(t--){
        cin >> n >> m;
        init_set();
        for(int i = 1; i <= m; i++){
            cin >> x >> y;
            union_set(x, y);
        }
        int ans = 0;
        for(int i = 1; i <= n; i++)       //统计有多少个集
            if(s[i] == i)
                ans++;
        cout << ans << endl;
    }
    return 0;
}
```

复杂度：在上述程序中，查找 find_set()、合并 union_set()的搜索深度是树的长度，复杂度都是 $O(n)$，性能比较差。下面介绍合并和查询的优化方法，优化之后，查找和合并的复杂度都小于 $O(\log_2 n)$。

2. 合并的优化

在合并元素 x 和 y 时先搜到它们的根结点，然后再合并这两个根结点，即把一个根结点的集改成另一个根结点。这两个根结点的高度不同，如果把高度较小的集合并到较大的集上，能减少树的高度。下面是优化后的代码，在初始化时用 height[i]定义元素 i 的高度，在合并时更改。

```
int height[maxn];
void init_set(){
   for(int i = 1; i <= maxn; i++){
```

```
            s[i] = i;
            height[i] = 0;                        //树的高度
        }
    }
    void union_set(int x, int y){                 //优化合并操作
        x = find_set(x);
        y = find_set(y);
        if (height[x] == height[y]) {
            height[x] = height[x] + 1;            //合并,树的高度加一
            s[y] = x;
        }
        else{                                     //把矮树并到高树上,高树的高度保持不变
            if (height[x] < height[y])  s[x] = y;
            else   s[y] = x;
        }
    }
```

3. 查询的优化——路径压缩

在上面的查询程序 find_set()中,查询元素 i 所属的集需要搜索路径找到根结点,返回的结果是根结点。这条搜索路径可能很长。如果在返回的时候顺便把 i 所属的集改成根结点,如图 5.5 所示,那么下次再搜的时候就能在 $O(1)$的时间内得到结果。

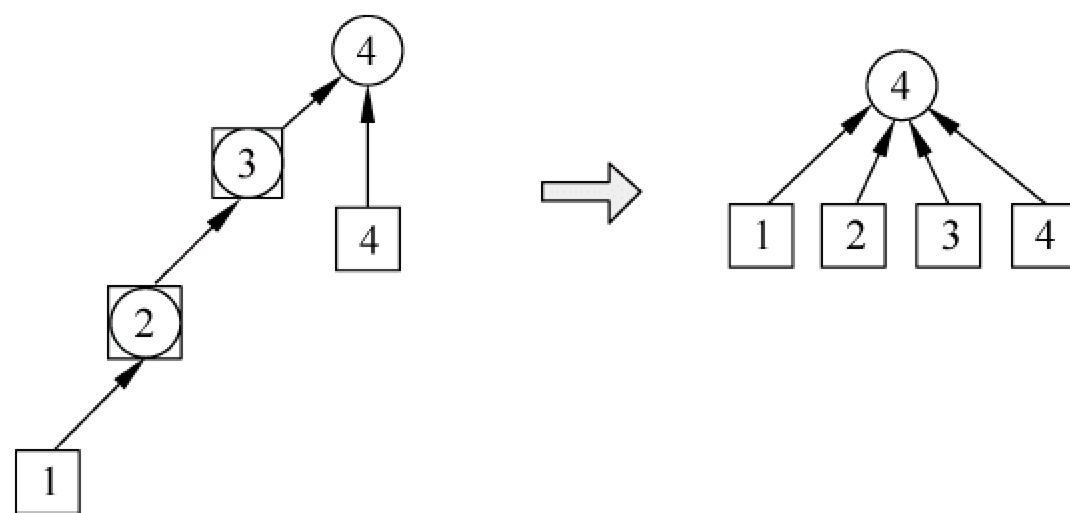


图 5.5 路径压缩

程序如下:

```
int find_set(int x){
    if(x != s[x]) s[x] = find_set(s[x]);         //路径压缩
    return s[x];
}
```

这个方法称为路径压缩,因为在递归过程中,整个搜索路径上的元素(从元素 i 到根结点的所有元素)所属的集都被改为根结点。路径压缩不仅优化了下次查询,而且优化了合并,因为在合并时也用到了查询。

上面的代码用递归实现,如果数据规模太大,担心爆栈,可以用下面的非递归代码:

```
int find_set(int x){
    int r = x;
    while (s[r] != r) r = s[r];                  //找到根结点
    int i = x, j;
    while(i != r){
```

```
            j = s[i];                          //用临时变量 j 记录
            s[i] = r;                          //把路径上元素的集改为根结点
            i = j;
        }
        return r;
    }
```

【习题】

poj 2524 “Ubiquitous Religions”,并查集简单题。

poj 1611 “The Suspects”,简单题。

poj 1703 “Find them, Catch them”。

poj 2236 “Wireless Network”。

poj 2492 “A Bug's Life”。

poj 1988 “Cube Stacking”。

poj 1182 “食物链”,经典题。

hdu 3635 “Dragon Balls”。

hdu 1856 “More is better”。

hdu 1272 “小希的迷宫”。

hdu 1325 “Is It A Tree”。

hdu 1198 “Farm Irrigation”。

hdu 2586 “How far away”,最近公共祖先,并查集+深搜。

hdu 6109 “数据分割”,并查集+启发式合并。

5.2 二 叉 树

树是非线性数据结构,它能很好地描述数据的层次关系。树形结构的现实场景很常见,例如,文件目录、书本的目录就是典型的树形结构。

二叉树是最常用的树形结构,特别适合程序设计,常常将一般的树转换成二叉树来处理。本节讲解二叉树的定义、遍历问题,以及二叉搜索树。

5.2.1 二叉树的存储

1. 二叉树的性质

二叉树的每个结点最多有两个子结点,分别是左孩子、右孩子,以它们为根的子树称为左子树、右子树。

二叉树的第 i 层最多有 2^{i-1} 个结点。如果每一层的结点数都是满的,称它为满二叉树。一个 n 层的满二叉树,结点数量一共有 2^n-1 个,可以依次编号为 $1,2,3,\cdots,2^n-1$。如果满二叉树只在最后一层有缺失,并且缺失的编号都在最后,那么称为完全二叉树。满二叉树和完全二叉树图示如图 5.6 所示。

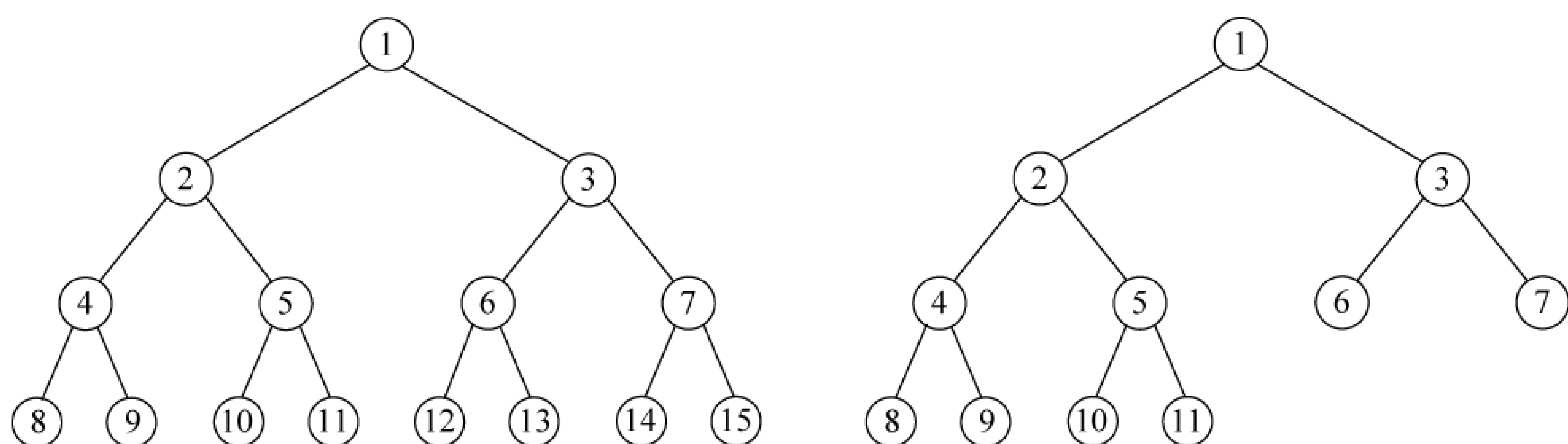


图 5.6　满二叉树和完全二叉树

完全二叉树非常容易操作。一棵结点数量为 k 的完全二叉树，设 1 号点为根结点，有以下性质：

(1) $i>1$ 的结点，其父结点是 $i/2$；

(2) 如果 $2i>k$，那么 i 没有孩子；如果 $2i+1>k$，那么 i 没有右孩子；

(3) 如果结点 i 有孩子，那么它的左孩子是 $2i$，右孩子是 $2i+1$。

2. 二叉树的存储结构

二叉树一般使用指针来实现，并指向左、右子结点。

```
struct node{
    int value;                          //结点的值
    node *l, *r;                        //指向左、右子结点
};
```

在新建一个 node 时，用 new 运算符动态申请内存。使用完毕后，应该用 delete 释放它，否则会内存泄漏。

二叉树也可以用数组来实现。特别是完全二叉树，用数组来表示父结点和子结点的关系非常简便。

5.2.2　二叉树的遍历



视频讲解

1. 宽度优先遍历

有时需要按层次一层层地遍历二叉树。例如在图 5.7 中需要按 *EBGADFICH* 的顺序访问，那么用宽度优先搜索是最合适的。用队列实现搜索的过程见本书 4.3 节。

2. 深度优先遍历

用深度优先搜索遍历二叉树，代码极其简单。

按深度搜索的顺序访问二叉树，对根(父)结点、左儿子、右儿子进行组合，有先(根)序遍历、中(根)序遍历、后(根)序遍历这 3 种访问顺序，这里默认左儿子在右儿子的前面。

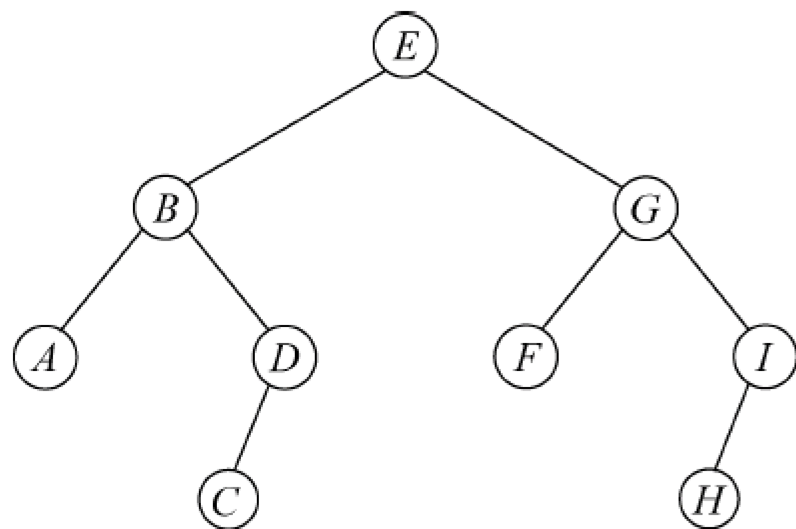


图 5.7　二叉树的遍历

(1) 先序遍历。即按父结点、左儿子、右儿子的顺序访问。在图 5.7 中，访问返回的顺序是 *EBADCGFIH*。

先序遍历的第1个结点是根。先序遍历的伪代码如下：

```
void preorder(node * root){
        cout << root -> value;                  //输出
        preorder(root -> l);                    //递归左子树
        preorder(root -> r);                    //递归右子树
}
```

(2) 中序遍历。按左儿子、父结点、右儿子的顺序访问。在图5.7中，访问返回的顺序是 *ABCDEFGHI*。读者可能注意到“*ABCDEFGHI*”刚好是字典序，这不是巧合，是因为图示的是一个二叉搜索树。在二叉搜索树中，中序遍历实现了排序功能，返回的结果是一个有序排列。中序遍历还有一个特征：如果已知根结点，那么在中序遍历的结果中，排在根结点左边的点都在左子树上，排在根结点右边的点都在右子树上。例如，*E* 是根，*E* 左边的“*ABCD*”在它的左子树上；再如，在子树“*ABCD*”上，*B* 是子树的根，那么“*A*”在它的左子树上，“*CD*”在它的右子树上。

(3) 后序遍历。按左儿子、右儿子、父结点的顺序访问。在图5.7中，访问返回的顺序是 *ACDBFHIGE*。后序遍历的最后一个结点是根。

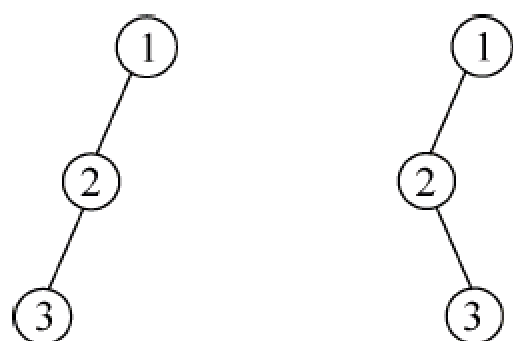


图5.8 “先序遍历+后序遍历”不能确定一棵树

如果已知某棵二叉树的3种遍历，可以把这棵树构造出来，即“中序遍历+先序遍历”或者“中序遍历+后序遍历”，都能确定一棵树。

但是，如果不知道中序遍历，只有“先序遍历+后序遍历”，不能确定一棵树。例如图5.8中两棵不同的二叉树，它们的先序遍历都是“1 2 3”，后序遍历都是“3 2 1”。

上述几种DFS遍历的实现见下面例题给出的代码。

hdu 1710 “Binary Tree Traversals”

输入二叉树的先序和中序遍历序列，求后序遍历。

(1) 输入样例。

先序：1 2 4 7 3 5 8 9 6

中序：4 7 2 1 8 5 9 3 6

(2) 输出样例。

后序：7 4 2 8 9 5 6 3 1

建树的过程如下：

(1) 先序遍历的第1个数是整棵树的根，例如样例中的“1”。知道了“1”是根，对照中序遍历，“1”左边的“4 7 2”都在根的左子树上，右边的“8 5 9 3 6”都在根的右子树上。

(2) 递归上述过程。例如，上面步骤得到的中序遍历的“4 7 2”，对照先序的第2个数是“2”，那么“2”是左子树的根，在中序遍历的“4 7 2”中，“2”左边的“4 7”都在以“2”为根的左子树上，等等。

图5.9所示为示意图，画线的数字是读取先序遍历逐一处理的当前步骤的根，方框内是中序遍历的部分数字。

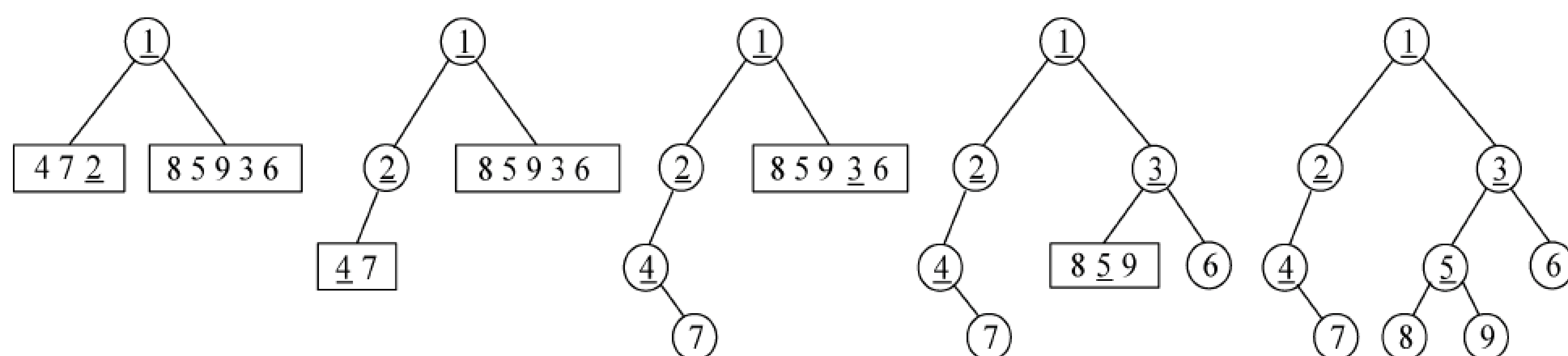


图 5.9 用先序遍历和中序遍历建二叉树

下面是 hdu 1710 的代码，其中 preorder()、inorder()、postorder()分别是先序遍历、中序遍历和后序遍历。可以看到，用 DFS 实现的二叉树遍历代码非常简单。

```
#include<bits/stdc++.h>
using namespace std;
const int N = 1010;
int pre[N], in[N], post[N];                    //先序、中序、后序
int k;
struct node{
    int value;
    node *l, *r;
    node(int value = 0, node *l = NULL, node *r = NULL):value(value), l(l), r(r){}
};
void buildtree(int l, int r, int &t, node* &root) {      //建树
    int flag = -1;
    for(int i = l; i<=r; i++)                   //先序的第 1 个数是根，找到对应的中序的位置
        if(in[i] == pre[t]){
            flag = i; break;
        }
    if(flag == -1) return;                      //结束
    root = new node(in[flag]);                  //新建结点
    t++;
    if(flag>l)  buildtree(l, flag - 1, t, root->l);
    if(flag<r)  buildtree(flag + 1, r, t, root->r);
}
void preorder(node *root){                      //求先序序列
    if(root != NULL){
        post[k++] = root->value;                //输出
        preorder(root->l);
        preorder(root->r);
    }
}
void inorder(node *root){                       //求中序序列
    if(root != NULL){
        inorder (root->l);
        post[k++] = root->value;                //输出
        inorder(root->r);
    }
}
void postorder(node *root){                     //求后序序列
```

```
    if(root != NULL){
        postorder(root ->l);
        postorder(root ->r);
        post[k++] = root ->value;            //输出
    }
}
void remove_tree(node * root){               //释放空间
    if(root == NULL) return;
    remove_tree(root->l);
    remove_tree(root->r);
    delete root;
}
int main(){
    int n;
    while(~scanf("%d", &n)){
        for(int i=1;i<=n;i++) scanf("%d", &pre[i]);
        for(int j=1;j<=n;j++) scanf("%d", &in[j]);
        node * root;
        int t = 1;
        buildtree(1, n, t, root);
        k = 0;                               //记录结点个数
        postorder(root);
        for(int i=0;i<k;i++) printf("%d%c",post[i],i==k-1?'\n':' ');
          //作为验证,这里可以用 preorder()和 inorder()检查先序和中序遍历
        remove_tree(root);
    }
    return 0;
}
```

代码中的 remove_tree()释放申请的空间,如果不释放,会内存泄漏,造成内存浪费。释放空间是标准的、正确的操作。不过,竞赛题目的代码很少,即使不释放空间,也不会出错;而且程序终止后,它申请的空间也会被系统收回。

5.2.5 节给出了用数组实现二叉树的例子。

5.2.3 二叉搜索树

BST(Binary Search Tree,二叉搜索树)是非常有用的数据结构,它的结构精巧、访问高效。BST 的特征如下:

(1) 每个元素有唯一的键值,这些键值能比较大小。通常把键值存放在 BST 的结点上。

(2) 任意一个结点的键值,比它左子树的所有结点的键值大,比它右子树的所有结点的键值小。也就是说,在 BST 上,以任意结点为根结点的一棵子树仍然是 BST。BST 是一棵有序的二叉树。可以推出,键值最大的结点没有右儿子,键值最小的结点没有左儿子。

图 5.10 是一棵二叉搜索树,用中序遍历可以得到它的有序排列。右图的虚线把每个结点隔开,很容易看出,结点正好按从小到大的顺序被虚线隔开了。有虚线的帮助,很容易理解后文介绍 Treap 树和 Splay 树时提到的“旋转”技术。

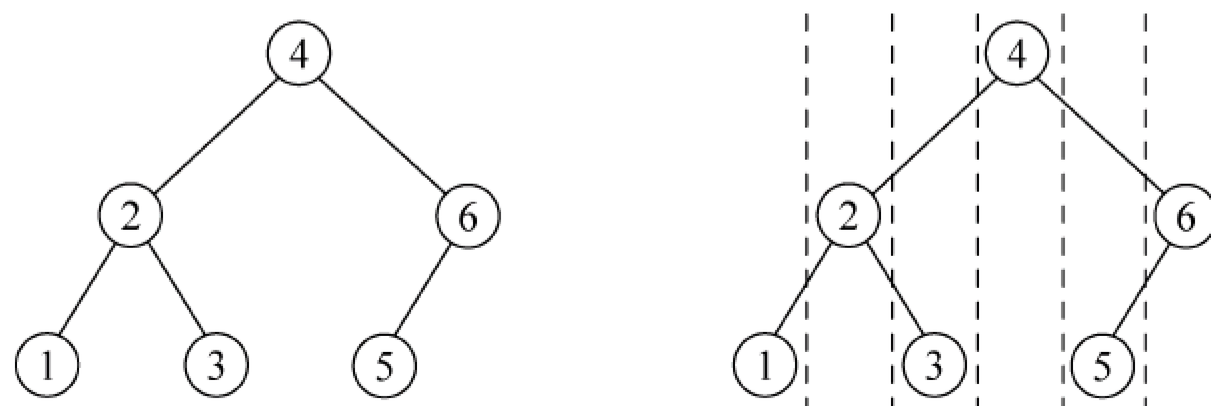


图 5.10　二叉搜索树

数据的基本操作是插入、查询、删除。给定一个数据序列，如何实现 BST？下面给出一种朴素的实现方法。

（1）建树和插入。以第 1 个数据 x 为根结点，逐个插入其他所有数据。插入过程从根结点开始，如果数据 y 比根结点 x 小，就往 x 的左子树上插，否则就往右子树上插；如果子树为空，就直接放到这个空位，如果非空，就与子树的值进行比较，再进入子树的下一层，直到找到一个空位置。新插入的数据肯定位于一个最底层的叶子结点，而不是插到中间某个结点上替代原来的数据。

从建树的过程可知，如果按给定序列的顺序进行插入，最后建成的 BST 是唯一的。形成的 BST 可能很好，也可能很坏。在最坏的情况下，例如一列有序整数{1，2，3，4，5，6，7}，按顺序插入，会全部插到右子树上；BST 退化成一个只包含右子树的链表，从根结点到最底层的叶子，深度是 n，导致访问一个结点的复杂度是 $O(n)$。在最好的情况下，例如序列{4，2，1，3，6，5，7}，得到的 BST 左、右子树是完全平衡的，深度是 $\log_2 n$，访问复杂度是 $O(\log_2 n)$。退化的 BST 和平衡 BST 如图 5.11 所示。

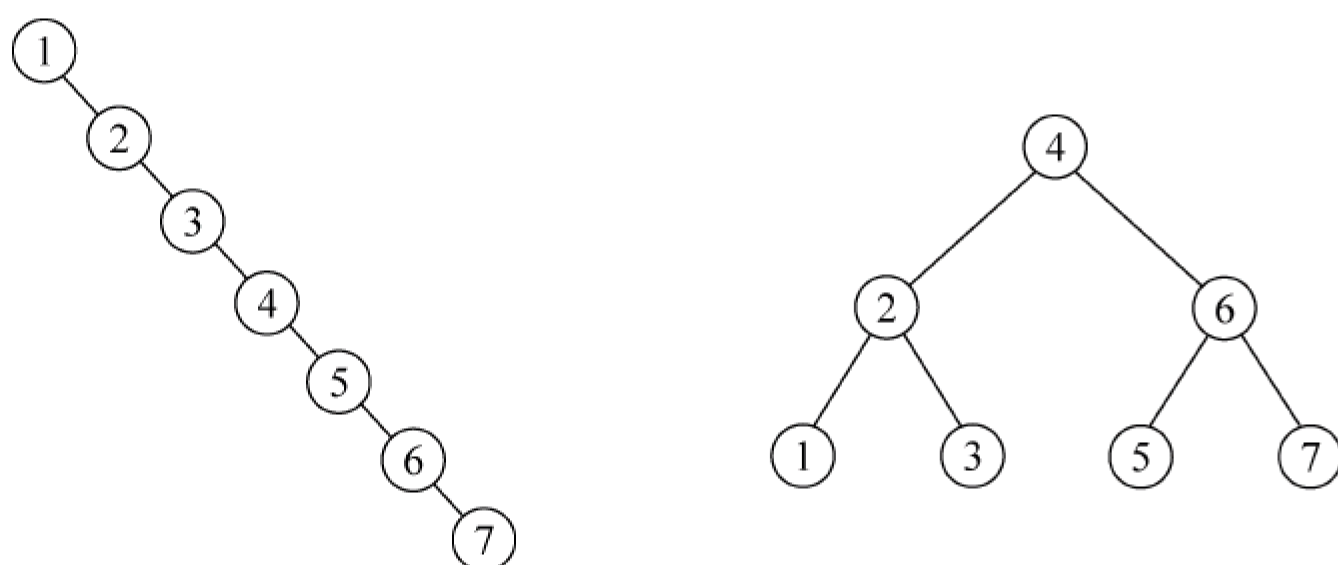


图 5.11　退化的 BST 和平衡 BST

（2）查询。建树过程实际上也是一个查询过程，所以查询仍然是从根结点开始的递归过程。访问的复杂度取决于 BST 的形态。

（3）删除。在删除一个结点 x 后，剩下的部分应该仍然是一个 BST。首先找到被删结点 x，如果 x 是最底层的叶子结点，直接删除；如果 x 只有左子树 L 或者只有右子树 R，直接删除 x，原位置由 L 或 R 代替。如果 x 左、右子树都有，情况就复杂了，此时，原来以 x 为根结点的子树需要重新建树。一种做法是，搜索 x 左子树中的最大元素 y，移动到 x 的位置，这相当于原来以 y 为根结点的子树，删除了 y，然后继续对 y 的左子树进行类似的操作，这也是一个递归的过程。删除操作的复杂度也取决于 BST 的形态。

（4）遍历。在 5.2.2 节中提到用中序遍历 BST，返回的是一个从小到大的排序。

根据上述过程可知，BST 的优劣取决于它是否为一个平衡的二叉树。所以，BST 有

关算法的主要功能是努力使它保持平衡。那么如何实现一个平衡的BST？由于无法提前安排元素的顺序（如果能一次读入所有元素，也能调整顺序，但是会大费周章，没有必要），所以只能在建树之后通过动态调整使它变得平衡。BST算法的区别就在于用什么办法调整。

BST算法有AVL树、红黑树、Splay树、Treap树、SBT树等。其中容易编程的有Splay树、Treap树等，也是算法竞赛中容易出的题目，本节后续讲解Treap树和Splay树。

BST是一个动态维护的有序数据集，用DFS对它进行中序遍历可以高效地输出字典序、查找第k大的数等。

STL与BST。STL中的set和map是用二叉搜索树（红黑树）实现的，检索和更新的复杂度是$O(\log_2 n)$。如果一个题目需要快速访问集合中的数据，可以用set或map实现，内容见本书第3章。

【习题】

hdu 3999 “The order of a Tree”，模拟BST的建树和访问。

hdu 3791 “二叉搜索树”，模拟BST。

poj 2418 “Hardwood Species”，用map快速处理字符串。

5.2.4 Treap树

首先研究一种比较简单的平衡二叉搜索树——Treap树。

Treap是一个合成词，把Tree和Heap各取一半组合而成。Treap是树和堆的结合，可以翻译成树堆。

二叉搜索树的每个结点有一个键值，除此之外，Treap树为每个结点人为添加了一个被称为优先级的权值。对于键值来说，这棵树是排序二叉树；对于优先级来说，这棵树是一个堆。堆的特征是：在这棵树的任意子树上，根结点的优先级最大。

1. Treap树的唯一性

Treap树的重要性质：令每个结点的优先级互不相等，那么整棵树的形态是唯一的，和元素的插入顺序没有关系。

下面用7个结点举例说明建树过程，其键值分别是$\{a,b,c,d,e,f,g\}$，优先级分别是$\{6,5,2,7,3,4,1\}$。图5.12(a)的纵向是优先级，横向是结点的键值；图5.12(b)按二叉搜索树的规则建了一棵树；图5.12(c)是结果。从这个图中可以看出Treap树的形态是唯一的。

2. Treap树的平衡问题

从图5.12可知，树的形态依赖于结点的优先级。那么如何配置每个结点的优先级，才能避免二叉树的形态退化成链表？最简单的方法是把每个结点的优先级进行随机赋值，那么生成的Treap树的形态也是随机的。这虽然不能保证每次生成的Treap树一定是平衡的，但是期望[①]的插入、删除、查找的时间复杂度都是$O(\log_2 n)$的。

① 关于期望的概念，见本书中的“8.4 概率和数学期望”。



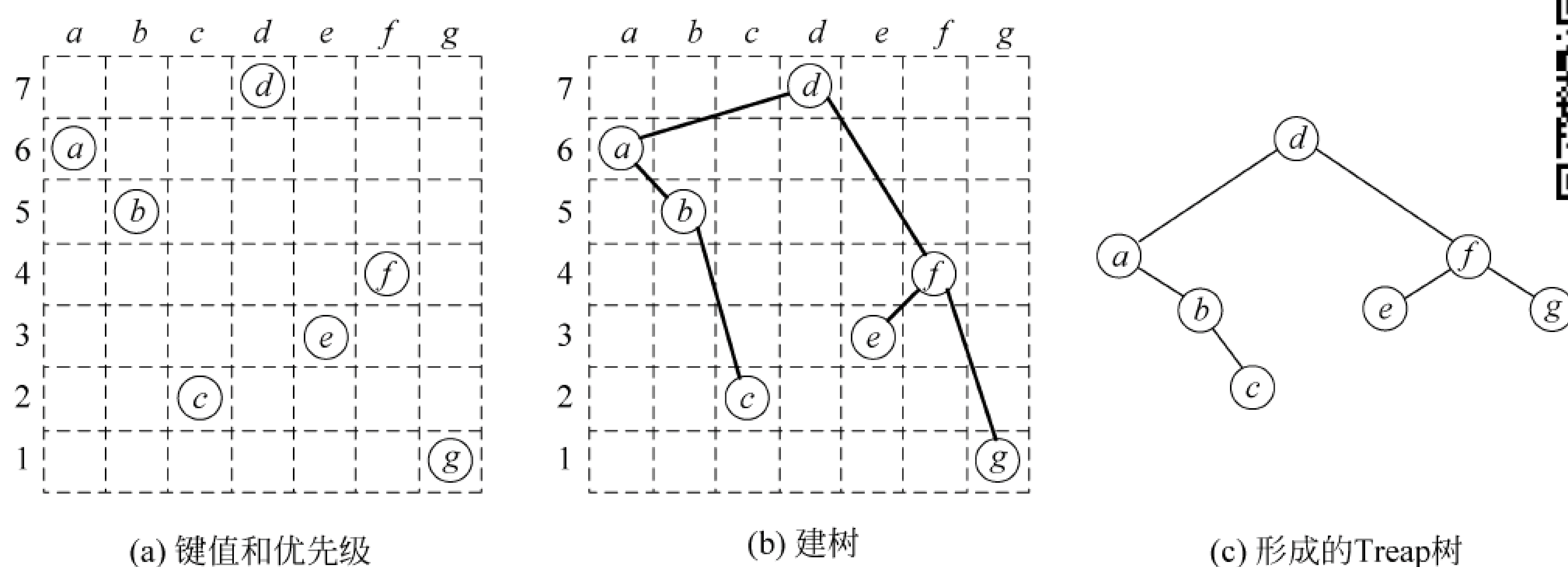


图 5.12　Treap 树的形态

了解了 Treap 树的概念，读者可以尝试自己完成建树的过程。在阅读下面的内容之前，不妨自己先试一试。

3. Treap 树的插入

如果预先知道所有结点的优先级，那么建树很简单，先按优先级排序，然后按优先级从高到低的顺序插入即可。例如在图 5.12 中，最高优先级的 d 第 1 个插入，是树根；第 2 优先级的 a 比 d 小，插到 d 的左子树上；第 3 优先级的 b 比 a 大，插到 a 的右子树……

不过，其实并不需要这么做。更简单的做法是每读入一个新结点，为它分配一个随机的优先级，插入到树中，在插入时动态调整树的结构，使它仍然是一棵 Treap 树。

把新结点 node 插入到 Treap 树的过程有以下两步：

(1) 用朴素的插入方法把 node 按键值大小插入到合适的子树上。

(2) 给 node 随机分配一个优先级，如果 node 的优先级违反了堆的性质，即它的优先级比父结点高，那么让 node 往上走，替代父结点，最后得到一个新的 Treap 树。

步骤(2)中的调整过程用到了一种技巧——旋转，包括左旋和右旋，如图 5.13 所示。

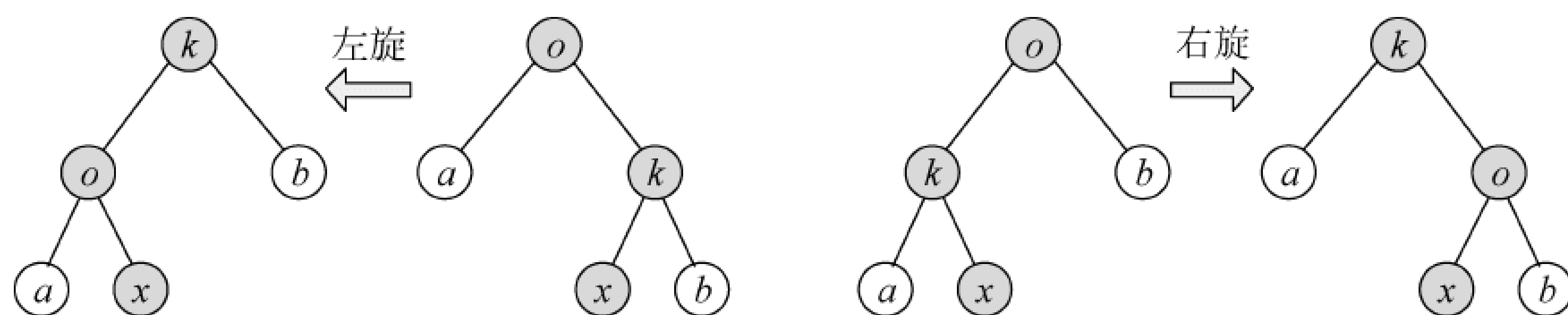


图 5.13　Treap 树的旋转(把 k 旋转到根)

旋转的代码如下，其中 son[0]是左儿子，son[1]是右儿子，代码中定义的结点名称和图 5.13 中的结点名称对应。

```
void rotate(Node * &o,int d){                    //d = 0,左旋转; d = 1,右旋
        Node  * k = o -> son[d ^1];              //d ^1 与 1 - d 等价,但是更快
        o -> son[d ^1] = k -> son[d];            //图中的 x
        k -> son[d] = o;
        o = k;                                   //返回新的根
    }
```

这里仍然以键值为$\{a,b,c,d,e,f,g\}$、优先级为$\{6,5,2,7,3,4,1\}$的 Treap 树为例，调整过程如下：图 5.14(a)是初始 Treap 树；图 5.14(b)插入 d 点，按朴素的插入方法插入到底部；图 5.14(c)中 d 的优先级比父结点 c 高，左旋，上升；图 5.14(d)中 d 的优先级比新的父结点 b 高，继续左旋，上升；图 5.14(e)中，d 再次左旋，上升，完成了新的 Treap 树。

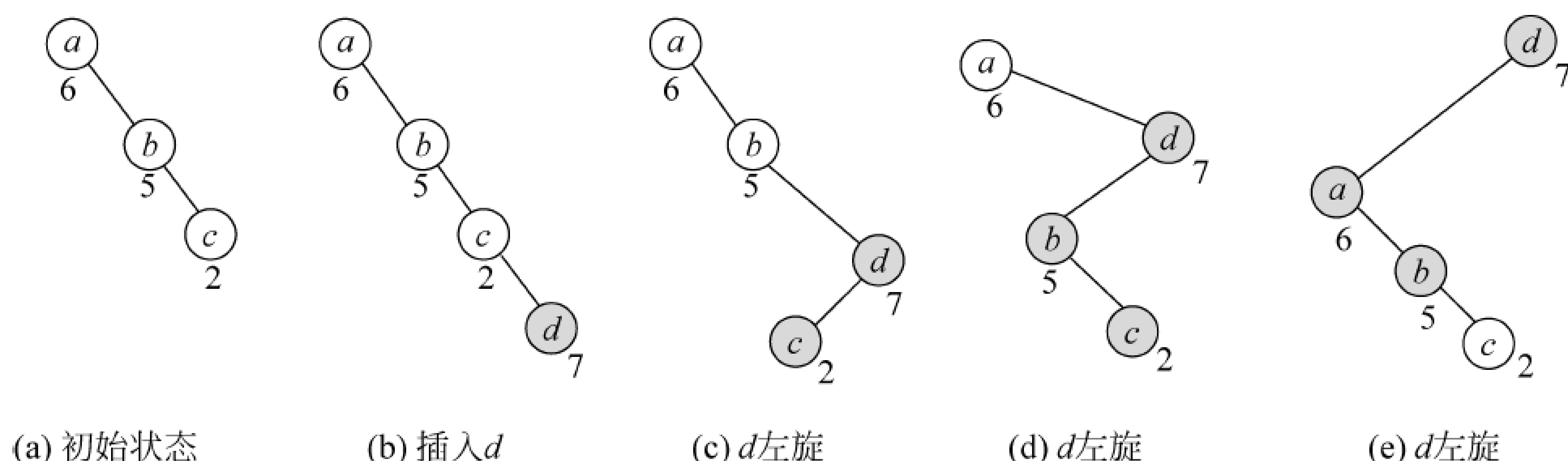


图 5.14　Treap 树的插入和调整

4. Treap 树的删除

如果待删除的结点 x 是叶子结点，直接删除。

如果待删除的结点 x 有两个子结点，那么找到优先级最大的子结点，把 x 向相反的方向旋转，也就是把 x 向树的下层调整，直到 x 被旋转到叶子结点，然后直接删除。

5. 分裂与合并问题

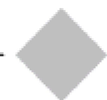
有时需要把一棵树分裂成两棵树，或者把两棵树合并成一棵树。Treap 树做这样的操作是比较烦琐的。读者可以用上面的例子尝试一下分裂和合并，例如在图 5.12(c)中，先把树分成$\{a,b\}$和$\{c,d,e,f,g\}$两棵树，然后再合并。注意在分裂和合并时仍然需要符合 Treap 树的规则。

5.2.5 节提到的 Splay 树做分裂和合并的操作非常简便。

6. Treap 与名次树问题

竞赛中与 Treap 有关的题目很多涉及名次树，例如：

> **hdu 4585 "Shaolin"**
>
> 少林寺的第 1 个和尚是方丈，作为功夫大师，他规定每个加入少林寺的年轻和尚要选一个老和尚来一场功夫战斗。每个和尚有一个独立的 id 和独立的战斗等级，新和尚可以选择跟他的战斗等级最接近的老和尚战斗。
>
> 方丈的 id 是 1，战斗等级是 10^9。他丢失了战斗记录，不过他记得和尚们加入少林寺的早晚顺序。请帮他恢复战斗记录。
>
> 输入：第 1 行是一个整数 n，$0<n\leqslant 100\,000$，和尚的人数，但不包括方丈本人。下面有 n 行，每行有两个整数 k、g，表示一个和尚的 id 和战斗等级，$0\leqslant k,g\leqslant 5\,000\,000$。和尚以升序排序，即按加入少林寺的时间排序。最后一行用 0 表示结束。
>
> 输出：按时间顺序给出战斗，打印出每场战斗中新和尚和老和尚的 id。



输入样例：
3
2 1
3 3
4 2
0
输出样例：
2 1
3 2
4 2

题意很简单，先对老和尚的等级排序，在加入一个新和尚时，找到等级最接近的老和尚，输出老和尚的 id。由于题目给的 n 比较大，因此总复杂度需要是 $O(n\log_2 n)$ 的。

此题有多种解法，这里给出两种解法——STL map、Treap 树。

1) STL map 代码

STL 的 map 和 set 都是用二叉搜索树实现的。这一题可以用 map 来做。

```
#include <bits/stdc++.h>
using namespace std;
map<int, int> mp;                          //it->first 是等级,it->second 是 id
int main(){
    int n;
    while (~scanf("%d",&n) && n){
        mp.clear();
        mp[1000000000] = 1;                //方丈 1,等级是 1 000 000 000
        while(n--){
            int id,g;
            scanf("%d%d",&id,&g);          //新和尚 id,等级是 g
            mp[g] = id;                    //新和尚进队
            int ans;
            map<int,int>::iterator it = mp.find(g);    //找到排好序的位置
            if (it == mp.begin())  ans = (++it)->second;
                else{
                map<int,int> :: iterator it2 = it;
                it2--; it++;               //等级接近的前后两个老和尚
                if (g-it2->first <= it->first-g)
                    ans = it2->second;
                else ans = it->second;
            }
            printf("%d %d\n",id,ans);
        }
    }
    return 0;
}
```

2）Treap 树代码

下面的 Treap 程序[①]给出了 Treap 树的常用操作：定义结点 struct Node、旋转 rotate()、插入 insert()、找第 k 大的数 kth()、查询某个数 find()。

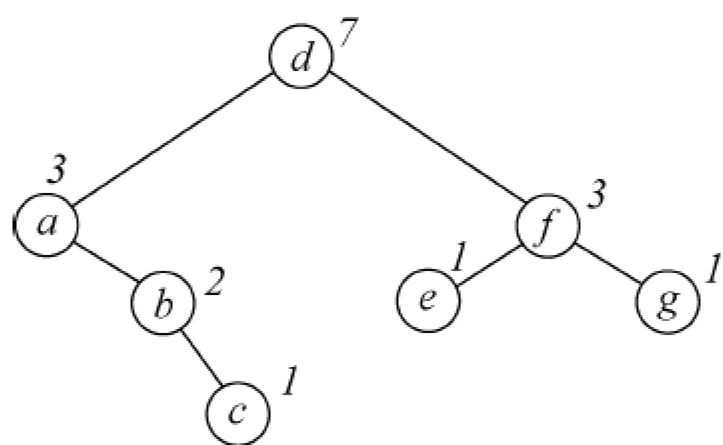


图 5.15　名次树

注意其中的 kth() 和 find()，它与名次树问题有关。名次树有两个功能：①找到第 k 大的元素；②查询元素 x 的名次，即 x 排名第几。这两个功能的实现借助于给结点增加的一个 size 值。一个结点的 size 值是以它为根的子树的结点总数量，例如图 5.15 所示的名次树。图中结点上标注的数字就是这个结点的 size。

下面的代码中给出了找第 k 大数的函数 kth() 以及查询元素名次的函数 find()，它们的复杂度都是 $O(\log_2 n)$ 的。

hdu 4585 的 Treap 代码（名次树）

```
#include <bits/stdc++.h>
using namespace std;
int id[5000000 + 5];
struct Node{
    int size;                          //以这个结点为根的子树的结点总数量，用于名次树
    int rank;                          //优先级
    int key;                           //键值
    Node *son[2];                      //son[0]是左儿子，son[1]是右儿子
    bool operator < (const Node &a)const{return rank < a.rank;}
    int cmp(int x)const{
        if(x == key) return -1;
        return x < key?0:1;
    }
    void update(){                     //更新 size
        size = 1;
        if(son[0]!= NULL) size +=  son[0]->size;
        if(son[1]!= NULL) size +=  son[1]->size;
        }
};
void rotate(Node* &o,int d){           //d = 0,左旋；d = 1,右旋
        Node *k = o->son[d^1];         //d^1 与 1-d 等价，但是更快
        o->son[d^1] = k->son[d];
        k->son[d] = o;
        o->update();
        k->update();
        o = k;
    }
void insert(Node* &o,int x){           //把 x 插入到树中
        if(o == NULL){
            o = new Node();
            o->son[0] = o->son[1] = NULL;
            o->rank = rand();
            o->key = x;
```

① 部分代码改编自《算法竞赛入门经典训练指南》，作者刘汝佳、陈锋，清华大学出版社，3.5.2 节，231 页。

```
            o->size=1;
        }
        else {
            int d=o->cmp(x);
            insert(o->son[d],x);
            o->update();
            if(o<o->son[d])
                rotate(o,d^1);
        }
    }
int kth(Node* o,int k){                    //返回第 k 大的数
        if(o==NULL||k<=0||k>o->size)
            return -1;
        int s=o->son[1]==NULL?0:o->son[1]->size;
        if(k==s+1) return o->key;
        else if(k<=s) return kth(o->son[1],k);
        else return kth(o->son[0],k-s-1);
    }
int find(Node* o,int k){  //返回元素 k 的名次
        if(o==NULL)
            return -1;
        int d=o->cmp(k);
        if(d==-1)
            return o->son[1]==NULL? 1: o->son[1]->size+1;
        else if(d==1) return find(o->son[d],k);
        else{
            int tmp=find(o->son[d],k);
            if(tmp==-1) return -1;
            else
                return o->son[1]==NULL? tmp+1 : tmp+1+o->son[1]->size;
        }
    }
int main(){
    int n;
    while(~scanf("%d",&n)&&n){
        srand(time(NULL));
        int k,g;
        scanf("%d%d",&k,&g);
        Node *root=new Node();
        root->son[0]=root->son[1]=NULL;
        root->rank=rand(); root->key=g; root->size=1;
        id[g]=k;
        printf("%d %d\n",k,1);
        for(int i=2;i<=n;i++){
            scanf("%d%d",&k,&g);
            id[g]=k;
            insert(root,g);
            int t=find(root,g);          //返回新和尚的名次
            int ans1,ans2,ans;
            ans1=kth(root,t-1);          //前一名的老和尚
            ans2=kth(root,t+1);          //后一名的老和尚
            if(ans1!=-1&&ans2!=-1)
                ans=ans1-g>=g-ans2 ? ans2:ans1;
            else if(ans1==-1) ans=ans2;
```

```
            else ans = ans1;
            printf("%d %d\n",k,id[ans]);
        }
    }
    return 0;
}
```

【习题】

poj 1442,名次树问题。

hdu 3726 “Graph and Queries”,离线算法+Treap 维护名次树。该题非常经典,是必做题。

5.2.5 Splay 树

Splay 树是一种 BST 树,它的查找、插入、删除、分割、合并等操作,复杂度都是 $O(\log_2 n)$ 的。它最大的特点是可以把某个结点往上旋转到指定位置,特别是可以旋转到根的位置,成为新的根结点。它有这样一种应用背景:如果需要经常查询和使用一个数,那么把它旋转到根结点,这样下次访问它,只需要查一次就找到了。

Splay 树有 Treap 树不具备的特点:①Splay 树允许把任意结点旋转到根,而 Treap 树不能,因为它的形态是固定的;②当需要分裂和合并时,Splay 树的操作非常简便。

下面介绍 Splay 操作,其中提根操作是核心。

1. 把结点旋转到根(提根)

Splay 树比 Treap 树的旋转操作的情况更多。

那么如何把一个结点 x 自底向上旋转到根?根据 x 的位置,有以下 3 种情况。

(1) x 的父结点就是根,只需要旋转一次。图 5.16 给出了 x 是根 c 的左儿子的情况,右儿子的情况与之类似。注意观察图中的中序遍历,即二叉搜索树的顺序“$a\ x\ b\ c\ d$”,保持不变。

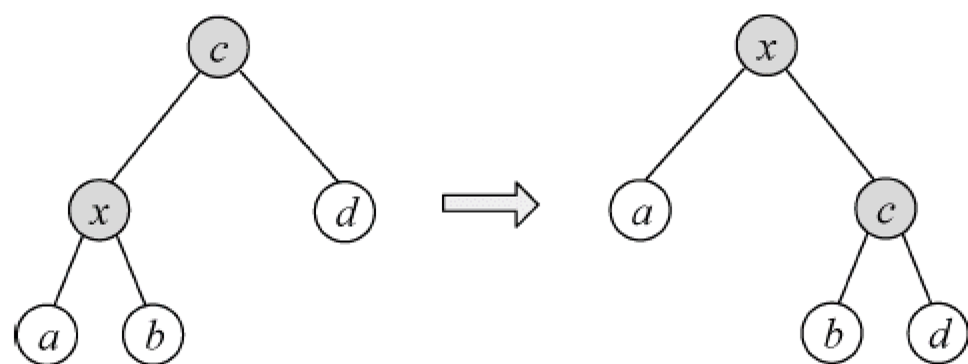


图 5.16　Splay 旋转情况 1

(2) x 的父结点不是根,x、x 的父结点、x 的父结点的父结点,三点共线。此时可以做两次单旋,即先旋转 x 的父结点,再旋转 x,如图 5.17 所示。

(3) x、x 的父结点、x 的父结点的父结点,三点不共线。把 x 按不同方向旋转两次,如图 5.18 所示。

按上述方法可以把任何深度的结点 x 旋转到根。

旋转一次的时间是个常数,那么把 x 从所在的深度提到根,总复杂度是多少?如果是平衡二叉树,最深的结点深度是 $O(\log_2 n)$,那么总复杂度就是 $O(\log_2 n)$。当然二叉树不一

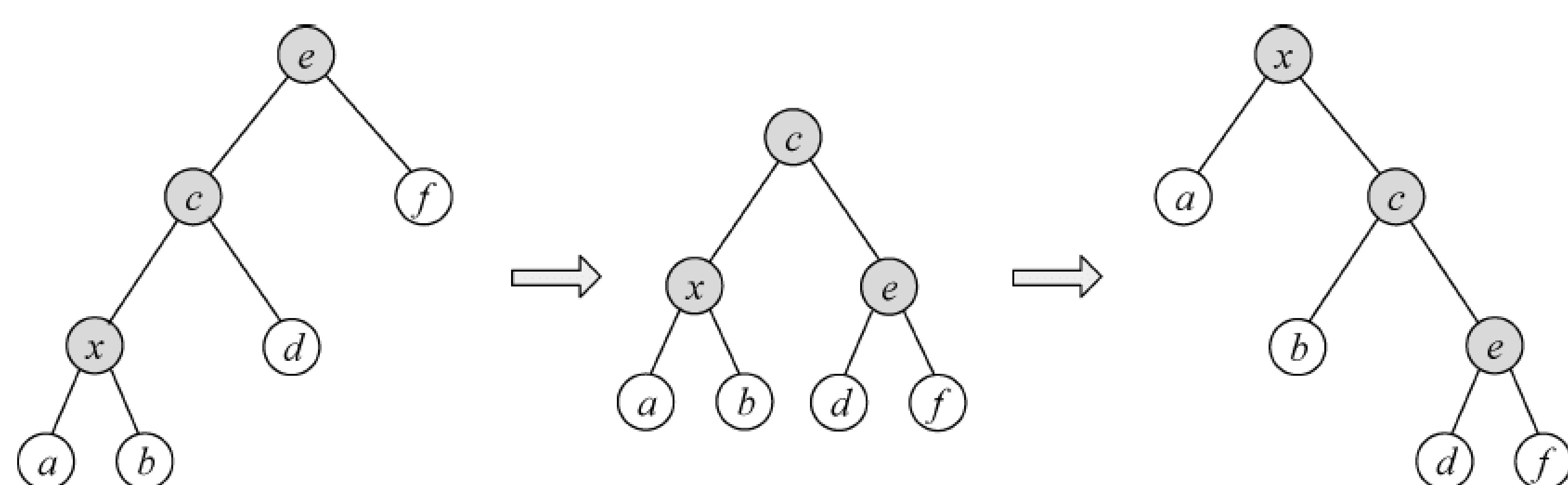


图 5.17　Splay 旋转情况 2

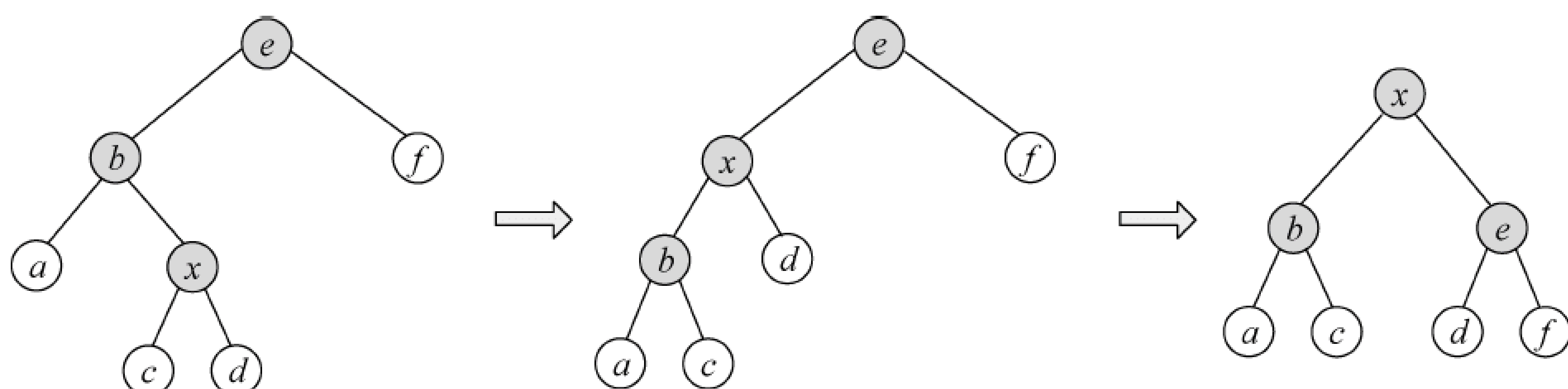


图 5.18　Splay 旋转情况 3

定是平衡的，不过在均摊意义上，可以把 Splay 提根操作的复杂度看成是 $O(\log_2 n)$的。这就是二叉树这种数据结构带来的优势。

下面的插入、分裂、合并，复杂度和提根的复杂度类似。

2. 插入

插入和普通二叉搜索树的方法一样。在插入之后，可以根据需要对新插入的结点做 Splay 操作。

3. 分裂

以第 k 小的数为界，把树分成两部分。先把第 k 小的元素旋转到树根，然后把它与右子树断开，就得到了两棵树。

4. 合并

可合并的两棵树，其中一棵树（设为 left）的所有元素应该小于另一棵树（设为 right）的所有元素。合并过程是先把 left 的最大元素 x 伸展到树根，此时树根 x 没有右子树，把 x 的右子树接到 right 的根，就完成了合并。

5. 删除

把待删除的结点旋转到根，删除它，然后合并左、右子树。

下面的例题给出了 Splay 树的编程细节。

---

hdu 1890 "Robotic Sort"

有 n 个数字，$1 \leqslant n \leqslant 100\,000$，用一个机械臂帮忙排序，其方法如图 5.19 所示。在左图中（图中的高度是数字大小），用机械臂夹住第 1 个数和最小的数，翻转，变成中图的样子，最小的数就处于第 1 个位置。然后对中图用同样的方法找第 2 小的数。继续这个过程直到结束。

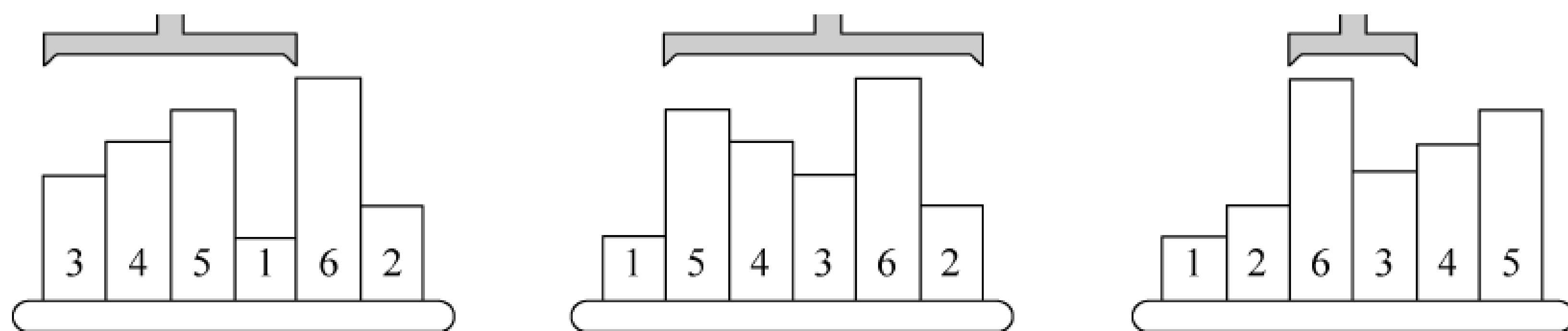


图 5.19　排序方法

输入一些数字，输出第 i 次翻转之前第 i 大的数的位置。

输入样例：3 4 5 1 6 2

输出样例：4 6 4 5 6 6

题目的基本操作是找到第 i 大的数，翻转它左边的数（不包括已经处理过的比它小的数），右边的数保持不变。如果用模拟法编程，复杂度约为 $O(n^3)$，会 TLE。本题需要 $O(n\log_2 n)$ 的方法。

注意，翻转有两种方法，这里以第 1 次翻转 3 4 5 1 为例：方法 1，直接翻转 3 4 5 1；方法 2，先把 1 挪到最左边，然后翻转 3 4 5。这两种方法的结果一样，在下面的 Splay 程序中适合用第 2 种方法。

本题的操作可以用 Splay 来模拟，利用了 Splay 树能把结点旋转到根的功能。

下面以第 1 个数的处理为例来说明过程。

（1）建树。把这个序列按初始位置建一个二叉搜索树。图 5.20(a) 是建树的结果，圆圈内是初始位置，圆圈旁边的数字是题目给出的序列。根据中序遍历，它是题目的样例3 4 5 1 6 2。建树的代码是 buildtree()。

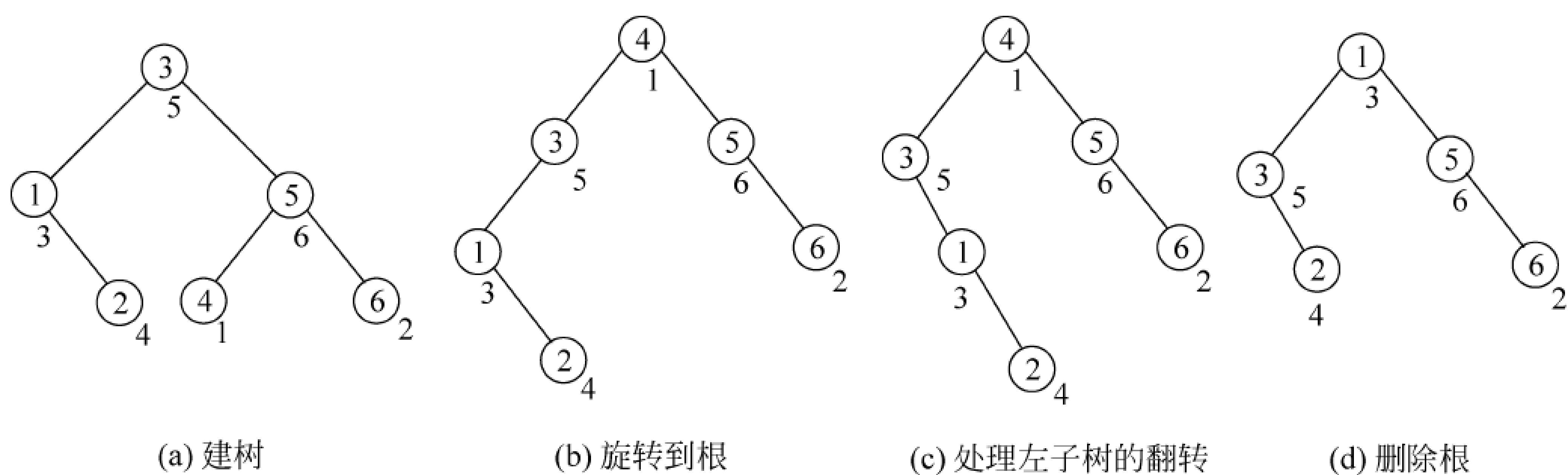
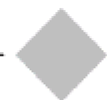


图 5.20　hdu 1890 题

（2）用 Splay 旋转到根。找到最小的数，用 Splay 把它旋转到根。其左子树的大小就是数列中排在它左边的数的个数，也就是题目的输出。其右边的数的顺序保持不变，左边的数需要模拟机械臂的翻转。旋转的代码是 splay()。

（3）翻转左子树。模拟题目中的机械臂翻转，但是，如果每次都完全翻转左子树，时间必然超时。这里从线段树①得到启发，用标记的方式记录翻转情况，减少直接操作的次数，

① 类似线段树的 lazy 操作，见本书中的“5.3.4 区间修改”。



等 Splay 操作的时候再处理。图(c)中只翻转了结点 3,对结点 3 做标记,而它的子树 1、2 保持不变。标记的代码是 update_rev()。注意,翻转会改变 BST 树的有序结构,所以本题并不是 Splay 树的裸题,只是用到了 Splay 树的旋转功能。在下面给出的代码中,如果去掉 update_rev(),就是纯粹的 Splay 代码。

(4) 删除根,即在树上删除最小数。在删除过程中,根据标记进行子树的翻转。最后的结果见图(d),这是去掉了最小数的树,第 1 次处理结束。删除根的代码是 del_root()。

下面是 hdu 1890 的代码。该代码中去掉 update_rev()、pushup()、pushdown(),就是纯的 Splay 代码。

```
#include<bits/stdc++.h>
using namespace std;
const int maxn = 100010;
int root;                                    //根
int rev[maxn],pre[maxn],size[maxn];
    //rev[i],标记 i 被翻转;pre[i],i 的父结点;size[i],i 的子树上结点的个数
int tree[maxn][2]; //记录树:tree[i][0],i 的左儿子;tree[i][1],i 的右儿子
struct node{
    int val,id;
    bool operator<(const node &A)const {   //用于 sort()排序
        if(val == A.val)return id<A.id;
        return val<A.val;
    }
}nodes[maxn];
void pushup(int x){                          //计算以 x 为根的子树包含多少子结点
    size[x] = size[tree[x][0]] + size[tree[x][1]] + 1;
}
void update_rev(int x){
    if(!x)return;
    swap(tree[x][0],tree[x][1]);             //翻转 x:交换左右儿子
    rev[x]^ = 1;                             //标记 x 被翻转
}
void pushdown(int x){                        //在做 Splay 时,根据本题的需要,处理机械臂翻转
    if(rev[x]){
        update_rev(tree[x][0]);
        update_rev(tree[x][1]);
        rev[x] = 0;
    }
}
void Rotate(int x,int c){                    //旋转,c = 0 为左旋,c = 1 为右旋
    int y = pre[x];
    pushdown(y);
    pushdown(x);
    tree[y][!c] = tree[x][c];
    pre[tree[x][c]] = y;
    if(pre[y])
        tree[pre[y]][tree[pre[y]][1] == y] = x;
    pre[x] = pre[y];
    tree[x][c] = y;
```

```
        pre[y] = x;
        pushup(y);
    }
    void splay(int x,int goal){
              //把结点 x 旋转为 goal 的儿子,如果 goal 是 0,则旋转到根
        pushdown(x);
        while(pre[x]!= goal){                 //一直旋转,直到 x 成为 goal 的儿子
            if(pre[pre[x]] == goal){          //情况(1):x 的父结点是根,单旋一次即可
                pushdown(pre[x]); pushdown(x);
                Rotate(x,tree[pre[x]][0] == x);
            }
            else{                             //x 的父结点不是根
                pushdown(pre[pre[x]]); pushdown(pre[x]); pushdown(x);
                int y = pre[x];
                int c = (tree[pre[y]][0] == y);
                if(tree[y][c] == x){          //情况(2):x、x 的父、x 父的父,不共线
                    Rotate(x,!c);
                    Rotate(x,c);
                }
                else{                         //情况(3):x、x 的父、x 父的父,共线
                    Rotate(y,c);
                    Rotate(x,c);
                }
            }
        }
        pushup(x);
        if(goal == 0) root = x;               //如果 goal 是 0,则将根结点更新为 x
    }
    int get_max(int x){
        pushdown(x);
        while(tree[x][1]){
            x = tree[x][1];
            pushdown(x);
        }
        return x;
    }
    void del_root(){                          //删除根结点
        if(tree[root][0] == 0){
            root = tree[root][1];
            pre[root] = 0;
        }
        else{
            int m = get_max(tree[root][0]);
            splay(m,root);
            tree[m][1] = tree[root][1];
            pre[tree[root][1]] = m;
            root = m;
            pre[root] = 0;
            pushup(root);
        }
    }
```

```
void newnode(int &x,int fa,int val){
    x = val;
    pre[x] = fa;
    size[x] = 1;
    rev[x] = 0;
    tree[x][0] = tree[x][1] = 0;
}
void buildtree(int &x,int l,int r,int fa){ //建树
    if(l > r) return;
    int mid = (l + r)>> 1;
    newnode(x,fa,mid);
    buildtree(tree[x][0],l,mid - 1,x);
    buildtree(tree[x][1],mid + 1,r,x);
    pushup(x);
}
void init(int n){
    root == 0;
    tree[root][0] = tree[root][1] = pre[root] = size[root] = 0;
    buildtree(root,1,n,0);
}
int main(){
    int n;
    while(~scanf("%d",&n) && n){
        init(n);
        for(int i = 1;i <= n;i++){
            scanf("%d",&nodes[i].val);   nodes[i].id = i;
        }
        sort(nodes + 1,nodes + n + 1);
        for(int i = 1;i < n;i++){
            splay(nodes[i].id,0);          //第 i 次翻转:把第 i 大的数旋到根
            update_rev(tree[root][0]);     //左子树需要翻转
            printf("%d ",i + size[tree[root][0]]);
              //i:第 i 次翻转;size:第 i 个被翻转数的左边的个数,就是它左子树的个数
            del_root();                    //删除第 i 次翻转的数,准备下一次翻转
        }
        printf("%d\n",n);
    }
    return 0;
}
```

读者可以在上面代码的基础上写出 Splay 树常见操作的代码,例如:

(1) 查找 x。执行 splay(x, 0),即把 x 旋转到根结点。

(2) 删除 x。先执行 splay(x, 0),把 x 旋转到根,然后用 del_root()删除它。

(3) 查找最大、最小、第 k 大的数。用中序遍历进行查找,查找后可以用 splay()把它旋转到根。

【习题】

hdu 1622,建二叉树;

hdu 3999,二叉树遍历;

hdu 3791,BST;

hdu 4453,Splay 基本题;

hdu 3726,离线处理+Splay,经典题。该题用 Treap 树也能做。

5.3 线 段 树

有这样一类 RMQ(Range Minimum/Maximum Query)问题,求区间最大值或最小值。设有长度为 n 的数列 $\{a_1,a_2,\cdots,a_n\}$,需要进行以下操作。

(1) 求最值:给定 $i,j \leqslant n$,求 $\{a_i,\cdots,a_j\}$ 区间内的最值。

(2) 修改元素:给定 k 和 x,把 a_k 改成 x。

如果用普通数组存储数列,上面两个操作中,求最值的复杂度是 $O(n)$,修改是 $O(1)$。如果有 m 次"修改元素+查询最值",那么总复杂度是 $O(mn)$。如果 m 和 n 比较大,例如 100 000 以上,那么整个程序的复杂度是 10^{10} 的数量级。这个复杂度在竞赛中是不可承受的。

除了 RMQ 问题以外,类似的还有求区间和问题。对于数列 $\{a_1,a_2,\cdots,a_n\}$,先更改某些数的值,然后给定 $i,j \leqslant n$,求 $\mathrm{sum}=a_i+\cdots+a_j$ 的区间和。对于单个更改或者求和,很容易写出 $O(n)$ 的算法;如果更改和询问的操作总次数是 m,那么整个程序的复杂度是 $O(mn)$。和 RMQ 一样,这样的复杂度也是不行的。

对于这类问题,有一种神奇的数据结构,能在 $O(m\log_2 n)$ 的时间内解决,这就是线段树。

5.3.1 线段树的概念



视频讲解

线段树是一种用于区间处理的数据结构,用二叉树来构造。

线段树是建立在线段(或者区间)基础上的树,树的每个结点代表一条线段[L,R]。图 5.21 所示为是线段[1,5]的线段树。

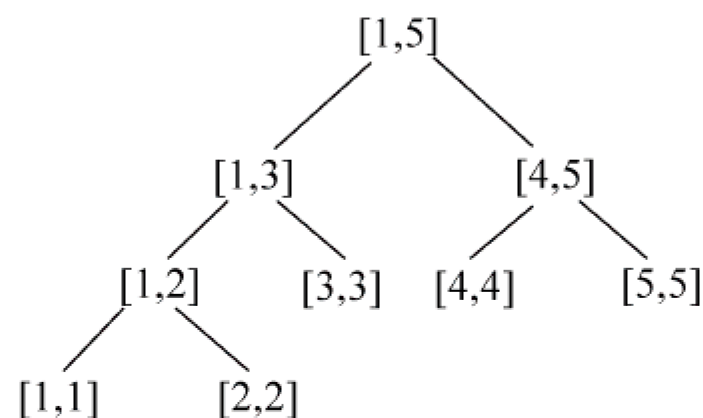


图 5.21 线段[1, 5]的线段树结构

考查每个线段$[L,R]$,L 是左子结点,R 是右子结点。

(1) $L=R$,说明这个结点只有一个点,它就是一个叶子结点。

(2) $L<R$,说明这个结点代表的不止一个点,它有两个儿子,左儿子代表的区间是$[L, M]$,右儿子代表的区间是$[M+1, R]$,其中 $M=(L+R)/2$。

线段树是二叉树,一个区间每次被折一半往下分,所以最多分 $\log_2 n$ 次就到达最低层。当需要查找一个点或者区间的时候,顺着结点往下找,最多 $\log_2 n$ 次就能找到。这就是线段树效率高的原因,使用了二叉树折半查找的方法。

回到 RMQ 问题,如果用线段树,"修改元素+查询最值"这两个操作分别可以在 $O(\log_2 n)$的时间内完成。如图 5.22 所示,查询{1,2,5,8,6,4,3}的最小值,其中每个结点上的数字是这棵子树的最小值。

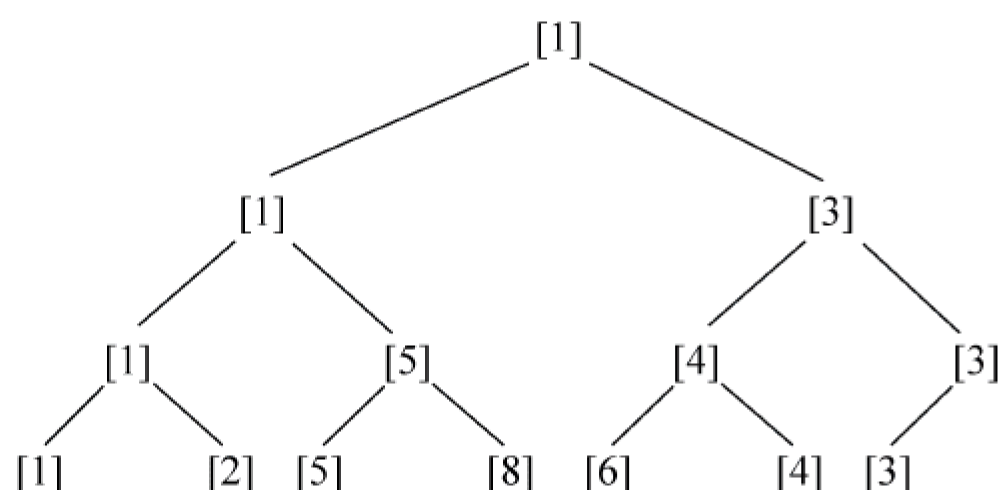


图 5.22　RMQ 问题(查询最小值)

如需修改元素,直接修改叶子结点上元素的值后从底往上更新线段树,操作次数也是 $O(\log_2 n)$。

m 次"修改+查询"的总复杂度是 $O(m\log_2 n\log_2 n)$。实际上,修改和查询可以同时做,所以总复杂度是 $O(m\log_2 n)$。这对规模 100 万的问题也能轻松解决。

5.3.2　点修改

首先讨论在线段树中每次只修改一个点的问题。

线段树如何构造?如何更新?如何查询?下面以 poj 2182 为例,引导出线段树的应用和编程细节。

> **poj 2182 "Lost Cows"**
>
> **题目描述**:有编号是 1～n 的 n 个数字,$2\leqslant n\leqslant 8000$,乱序排列,顺序是未知的。对于每个位置的数字,知道排在它前面比它小的数字有多少个。求这个乱序数列的顺序。

例如有 5 个数,已知每个数字前面比它小的数的个数,分别是:

pre[]:　0 1 2 1 0

可以求得这个乱序排列是:

ans[]:　2 4 5 3 1

本题是"简单题",用线段树或者树状数组实现。

在讲解后续内容之前,这里先用暴力法实现,思路是从后往前处理 pre[]:

(1) pre[5]=0,表示 ans[5]前面比它小的数有 0 个,即 ans[5]是最小的,在 1～5 这几个编号中 1 最小,所以 ans[5]=1。ans[]的前 4 个编号不再包括 1,剩下 2～5 这几个编号。

(2) pre[4]=1,在剩下的 2～5 这几个编号中,编号 3 是第 2 大的,所以 ans[4]=3。

(3) pre[3]=2,在剩下的 2、4、5 这几个编号中,编号 5 是第 3 大的,所以 ans[3]=5。

(4) pre[2]=1,在剩下的 2、4 这两个编号中,编号 4 是第 2 大的,所以 ans[2]=4。

(5) pre[1]=0,剩下的编号 2,ans[1]=2。

概括以上步骤,在每一步,剩下的编号中第 pre[n]+1 大的编号就是 ans[n]。

用暴力的方法,从 pre[]末尾往前计算,每处理一头牛后,需要把剩下的牛重新排名,重新排名的计算时间是 $O(n)$;在重新排名时,可以顺便做下一次的查找,所以不需要另外算查找的时间。一共有 n 头牛,总复杂度是 $O(n^2)$。本题的数据规模 n 不大,只有 8000,用暴力的方法也能通过。

在下面的代码中，pre[]是输入的 1～n 个数字，例如{0 1 2 1 0}；ans[]是答案，例如{2 4 5 3 1}；num[]记录被处理过的数字，被处理后的数字置为－1。例如 num[]的初始值是{1 2 3 4 5}，得到 ans[5]＝1 后，num[]更新为{－1 2 3 4 5}，这里用－1 表示 1 这个数字已经用过了。

解题的关键是，在剩下的编号中，第 pre[n]＋1 个数字就是 ans[n]。

下面是代码。

poj 2182 的暴力法代码

```
#include <stdio.h>
const int Max = 8005;
int main(){
    int n, i, j, k;
    int pre[Max], ans[Max], num[Max];      //数组的第 0 个都不用，从第 1 个开始用
    scanf("%d", &n);
    pre[1] = 0;
    for(i = 1; i <= n; i++)        num[i] = i;
    for(i = 2; i <= n; i ++)      scanf("%d", &pre[i]);
    for(i = n; i >= 1; i -- ) {                //从后往前处理数列
        k = 0;
        for(j = 1; j <= n; j++)                //查找 num[]中未处理的第 pre[i] + 1 大的数
            if(num[j] != -1) {
                k++;
                if(k == pre[i] + 1) {          //找到了
                    ans[i] = num[j];   //num[]中剩下的第 pre[i] + 1 个数就是 ans[i]
                    num[j] = -1;
                    break;
                }
            }
    }
    for(i = 1; i <= n; i++)    printf("%d\n", ans[i]);
    return 0;
}
```

当 n 更大时，$O(n^2)$会 TLE，必须用更优的算法。问题的关键是，如何高效地对剩下的牛重新排名。

这里引入高级数据结构“线段树”，可以在 $O(\log_2 n)$的时间内完成一次重新排名。下面说明其要点。

1. 用二叉树建立线段树

用二叉树的方法，把牛分成不同的组。在图 5.23 中，叶子结点内的数字是牛的编号，其他结点是牛的编号范围。例如根结点，包含 5 头牛，它的左子结点有 3 头牛，右子结点有两头牛。

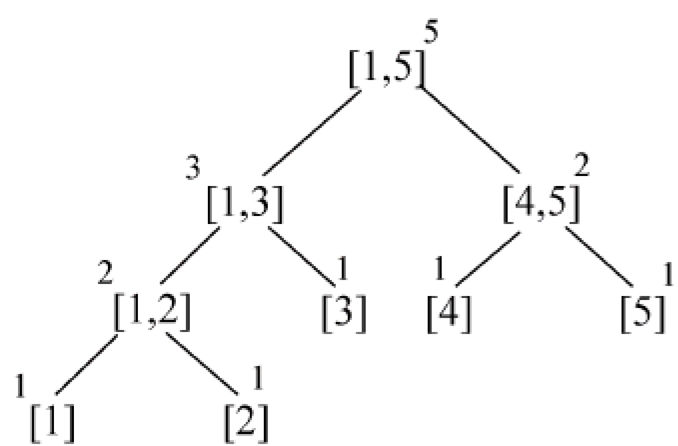


图 5.23　初始线段树

2. 存储空间

如果牛有 n 头，这个二叉树的结点总数在编程时为

4*n*。请读者自己思考为什么是 4*n*(在后面的程序注释中有答案)。

3. 查询和更新

(1) 第 1 次处理 pre[5]=0,即找对应的第 1 头牛,如图 5.24(a)所示。步骤是从根结点开始,逐步找到左下角,即编号为 1 的结点,得到 ans[5]=1。在这个过程中,更新经过的每个结点,即把这个结点剩下的牛的数量减一。一共需要更新 4 个结点。左下角结点已经减到 0,表示后面的计算需要排除它。

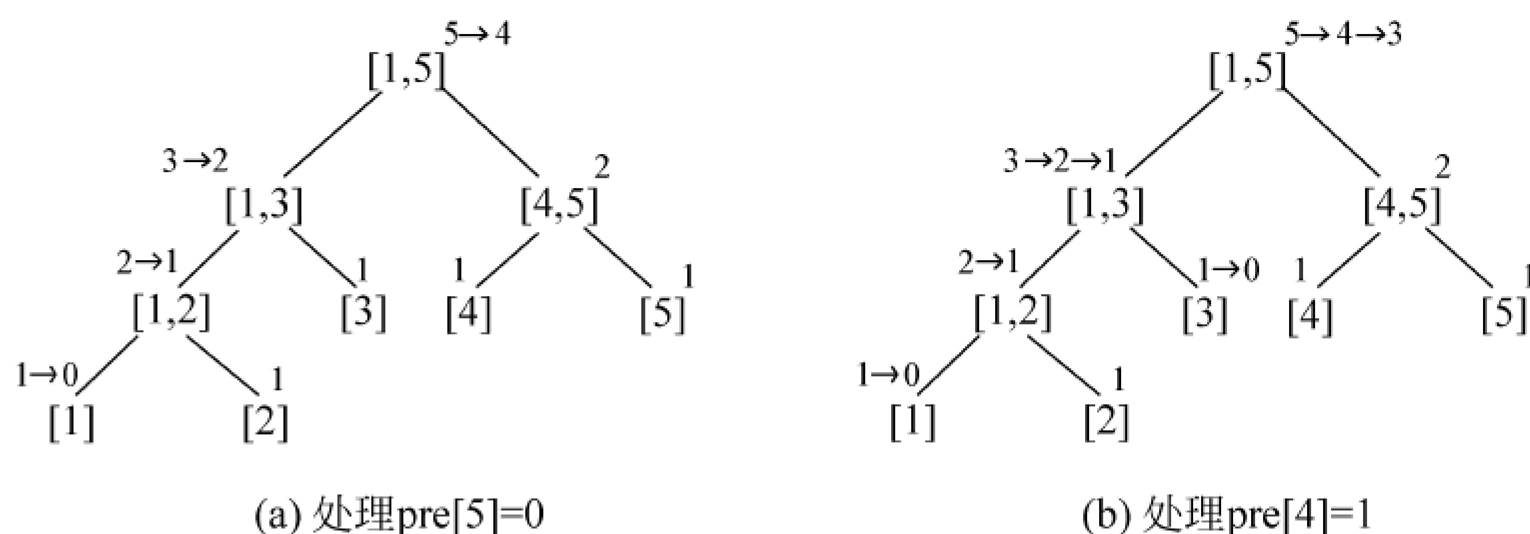


图 5.24　线段树的查询和更新

(2) 第 2 次处理 pre[4]=1,即找剩下的第 2 头牛,如图 5.24(b)所示。步骤是从根结点开始,逐步找到左边第 3 个结点,得到 ans[5]=3。更新经过的每个结点,一共更新 3 个结点。

(3) 依次处理,直到结束。

4. 复杂度

每次处理,从二叉树的根结点开始到最下一层,最多需要更新 $\log_2 4n$ 个结点,复杂度是 $O(\log_2 n)$;一共有 *n* 头牛需要处理,总复杂度是 $O(n\log_2 n)$。在暴力法中,每次需要查询和更新 *n* 个序列中的每个数,复杂度为 $O(n)$。线段树把 *n* 个数按二叉树进行分组,每次更新有关的结点时,这个结点下面的所有子结点都隐含被更新了,从而大大地减少了处理次数。

下面给出 poj 2182 的线段树代码。

poj 2182 "用结构体实现线段树"

```
#include <stdio.h>
using namespace std;
const int Max = 10000;
struct{
    int l, r, len;                    //用 len 存储这个区间的数字个数,即这个结点下牛的数量
}tree[4 * Max];                       //这里是 4 倍,因为线段树的空间需要
int pre[Max], ans[Max];
void BuildTree(int left, int right, int u){ //建树
    tree[u].l = left;
    tree[u].r = right;
    tree[u].len = right - left + 1;       //更新结点 u 的值
     if(left == right)
        return;
    BuildTree(left, (left + right)>> 1, u << 1);             //递归左子树
    BuildTree(((left + right)>> 1) + 1, right, (u << 1) + 1);  //递归右子树
```

```
}
int query(int u, int num){                  //查询 + 维护,所求值为当前区间中左起第 num 个元素
    tree[u].len --;                         //对访问到的区间维护 len,即把这个结点上牛的数量减一
    if(tree[u].l == tree[u].r)
        return tree[u].l;
//情况 1:左子区间内牛的个数不够,则查询右子区间中左起第 num - tree[u<<1].len 个元素
    if(tree[u<<1].len < num)
        return query((u<<1) + 1, num - tree[u<<1].len);
//情况 2:左子区间内牛的个数足够,依旧查询左子区间中左起第 num 个元素
    if(tree[u<<1].len >= num)
        return query(u<<1, num);
}
int main(){
    int n, i;
    scanf("%d", &n);
    pre[1] = 0;
    for(i = 2; i <= n; i ++)
        scanf("%d", &pre[i]);
    BuildTree(1, n, 1);
    for(i = n; i >= 1; i --)                //从后往前推断出每次最后一个数字
        ans[i] = query(1, pre[i] + 1);
    for(i = 1; i <= n; i ++)
        printf("%d\n", ans[i]);
    return 0;
}
```

5. 用完全二叉树实现线段树

在上面的例子中,线段树是一棵普通的二叉树,操作起来比较麻烦。其实可以用完全二叉树的结构来实现,编程更加简单,如图 5.25 所示。

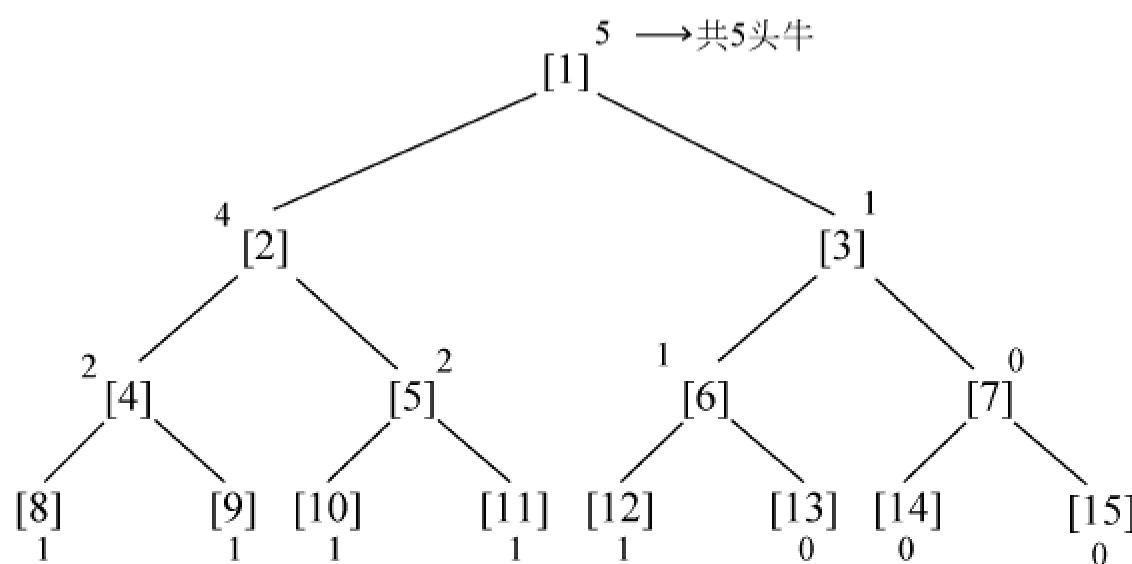
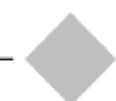


图 5.25 用完全二叉树建线段树

该图中的最后一行是牛的编号,例如[8]对应 1 号牛,[9]对应 2 号牛,等等。一共有 5 头牛。

在使用完全二叉树时,最后一层会存在"空叶子"。同样给空叶子按顺序编号,在遍历线段树时根据判断条件跳过这些"空叶子"就好了。用完全二叉树的方式存储线段树能提高插入线段和搜索时的效率。父结点 p 的左、右子结点分别是 $p*2$、$p*2+1$,用这样的索引方式检索 p 的左、右子树比用指针快。



poj 2182“用完全二叉树实现线段树”

```
#include <stdio.h>
#include <math.h>
const int Max = 10000;
int pre[Max] = {0}, tree[4 * Max] = {0},ans[Max] = {0};
//tree 是用数组实现的满二叉树.从图 5.25 可以知道,需要 4 倍大的空间
void BuildTree(int n,int last_left){        //用完全二叉树建一个线段树
    int i;
    for(i = last_left;i < last_left + n;i++)
                //给二叉树的最后一行赋值,左边 n 个结点是 n 头牛
        tree[i] = 1;
    while(last_left != 1) {      //从二叉树的最后一行倒推到根结点,根结点的值是牛的总数
        for(i = last_left/2; i < last_left; i++)
            tree[i] = tree[i * 2] + tree[i * 2 + 1];
        last_left = last_left/2;
    }
}
int query(int u, int num,int last_left){
    //查询 + 维护,关键的一点是所求值为当前区间中左起第 num 个元素
    tree[u]--;                          //对访问到的区间维护剩下的牛的个数
    if(tree[u] == 0 && u >= last_left)
        return u;
//情况 1:左子区间的数字个数不够,则查询右子区间中左起第 num - tree[u<<1]个元素
    if(tree[u<<1] < num)
        return query((u<<1) + 1, num - tree[u<<1], last_left);
//情况 2:左子区间的数字个数足够,依旧查询左子区间中左起第 num 个元素
    if(tree[u<<1] >= num)
        return query(u<<1, num,last_left);
}
int main(){
    int n, last_left,i;
    scanf("%d", &n);
    pre[1] = 0;
    last_left = 1<<(int(log(n)/log(2)) + 1);
//二叉树最后一行的最左边一个.计算方法是找离 n 最近的 2 的指数,例如 3->4, 4->4, 5->8
    for(i = 2; i <= n; i ++)
        scanf("%d", &pre[i]);
    BuildTree(n,last_left);
    for(i = n; i >= 1; i--)              //从后往前推断出每次最后一个数字
        ans[i] = query(1, pre[i] + 1,last_left) - last_left + 1;
    for(i = 1; i <= n; i ++)
        printf("%d\n", ans[i]);
    return 0;
}
```

5.3.3　离散化

建二叉树是线段树的基本操作,但是二叉树的大小并不是无限制的,例如规模 10 000 000

以上的二叉树会超过允许的存储空间。在竞赛中如果出现结点规模这样大的题目，当然不能在程序中建这么大的二叉树，此时需要用“离散化”这种小技巧来解决。

离散化就是把原有的大二叉树压缩成小二叉树，但是压缩前后子区间的关系不变。

例如一块宣传栏，横向长度的刻度标记为 1 到 10，贴 4 张不同颜色的海报，它们的宽度和宣传栏等宽，长度分别是[1,3]、[2,5]、[3,8]、[3,10]，并且用后者覆盖前者，问最后能看见几种颜色的海报。

离散化步骤如下：

(1) 提取这 4 张海报的 8 个端点：1 3 2 5 3 8 3 10

(2) 排序并且删除相同的端点，得到：1 2 3 5 8 10

(3) 把原线段的 8 个端点映射到新的线段上：

1	2	3	5	8	10
↓	↓	↓	↓	↓	↓
1	2	3	4	5	6

新的 4 个海报为[1,3]、[2,4]、[3,5]、[3,6]，覆盖关系没有改变。新的宣传栏长度是 1 到 6，即宣传栏的长度从 10 压缩到 6。

离散化的压缩比是很可观的。例如原线段树的区间长度是 10 000 000，而其中真正用到的子区间是 100 000，那么子区间的端点最多有 2×100 000 个。经过离散化压缩后，新的线段树区间是 200 000，压缩率是 200 000/10 000 000=2%。

【习题】

poj 2528，题目中宣传栏的长度是 10 000 000。

5.3.4 区间修改

上面的例子都是只修改线段树上的某个点。区间修改是更复杂的问题。给定 n 个元素 $\{a_1, a_2, \cdots, a_n\}$，进行以下操作：

加：给定 $i, j \leqslant n$，把 $\{a_i, \cdots, a_j\}$ 区间内的值全部加 v。

查询：给定 $L, R \leqslant n$，计算 $\{a_L, \cdots, a_R\}$ 的区间和。

下面以 poj 3468 为例来讲解区间修改问题。

> **poj 3468 “A Simple Problem with Integers”**
>
> 给出 N 个数，进行 Q 个操作，$1 \leqslant N, Q \leqslant 100\ 000$。有两种操作：
>
> “C a b c”，对区间$[a,b]$的每个数字加 c；
>
> “Q a b”，查询区间$[a,b]$的数字和。
>
> 输入：N，Q，以及 N 个数字，Q 个操作；
>
> 输出：对每个查询操作，输出结果。

如果用暴力方法，直接对这 n 个数进行操作，那么每个 C 操作和 Q 操作都是 $O(n)$ 的，一共有 Q 次操作，总复杂度是 $O(n^2)$。

如果用前面的修改线段树点的方法，在做 C 操作时，对区间里的数一个一个进行修改，

一个数的修改是 $O(\log_2 n)$ 的，区间修改合起来是 $O(n\log_2 n)$，Q 次操作的总复杂度是 $O(n^2\log_2 n)$，比暴力法还要差。

lazy-tag 方法。此时可以采用一种“懒惰(lazy)”的做法。当修改的是一个整块区间时，只对这个线段区间进行整体上的修改，其内部每个元素的内容先不做修改，只有当这部分线段的一致性被破坏时才把变化值传递给子区间。那么，每次区间修改的复杂度是 $O(\log_2 n)$，一共有 Q 次操作，总复杂度是 $O(n\log_2 n)$。做 lazy 操作的子区间，需要记录状态(tag)，在下面的代码中用 add[]实现。

视频讲解

下面描述具体步骤。

(1) 初始化时建树。以区间[1, 10]为例建树，图 5.26 所示为结果。在最后的叶子上是 1～10 这 10 个数字。图中最底层有很多叶子是空的。每个结点右上角的数字是以它为根结点的这棵子树的区间和。

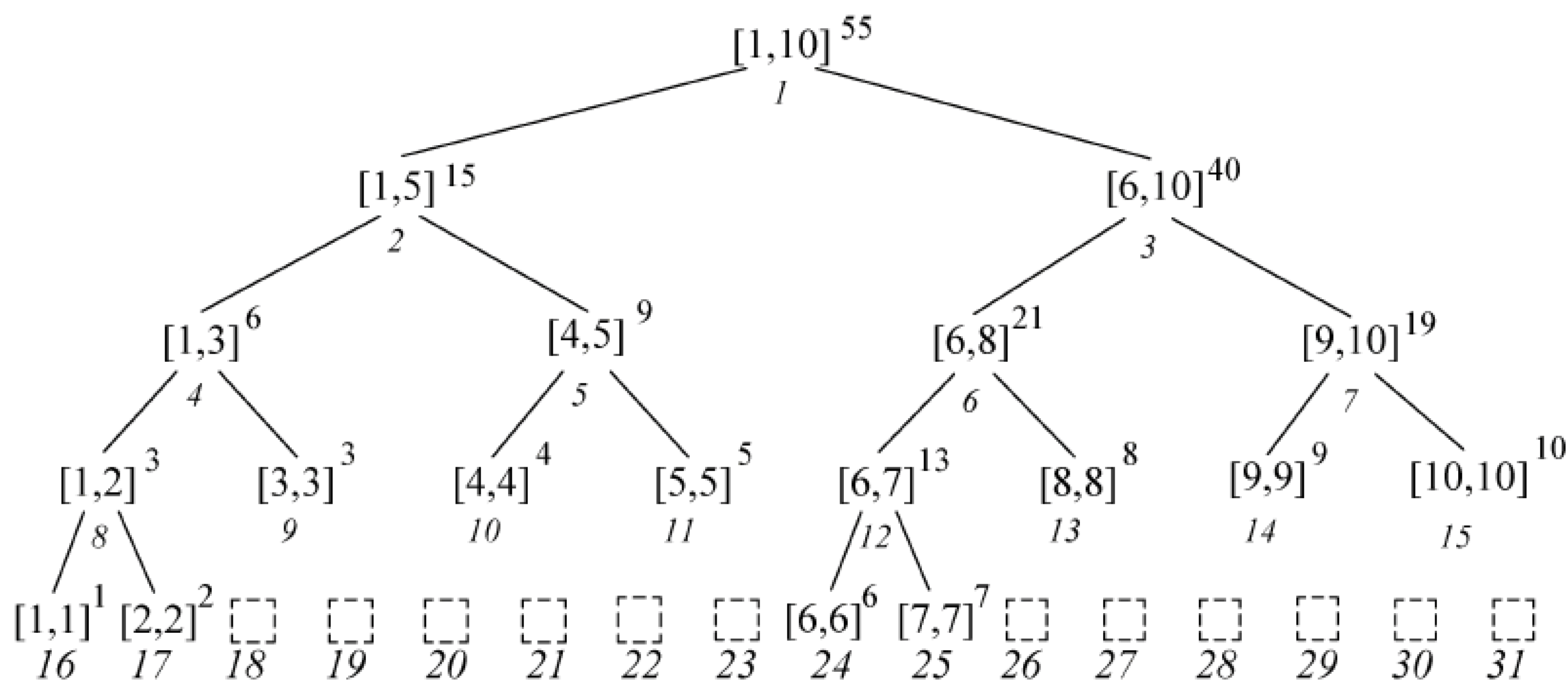


图 5.26　初始化建树

(2) “C a b c”操作。例如“C 3 6 3”，在[3, 6]区间内，把每个元素加 3。从根结点开始，用递归在子树中找区间[3, 6]，有两种情况：[3, 6]与子区间交错、[3, 6]包含子区间。例如子区间[1, 5]和[6, 10]都与[3,6]交错，需要继续深入更底层子区间。在子区间[4, 5]，它被[3, 6]包含，那么根据 lazy 原理，把这个子区间进行整体修改，不继续深入，它下一层的[4, 4]和[5, 5]的区间和不用修改。图 5.27 所示为结果。部分结点的区间和发生了改变，见右上角。

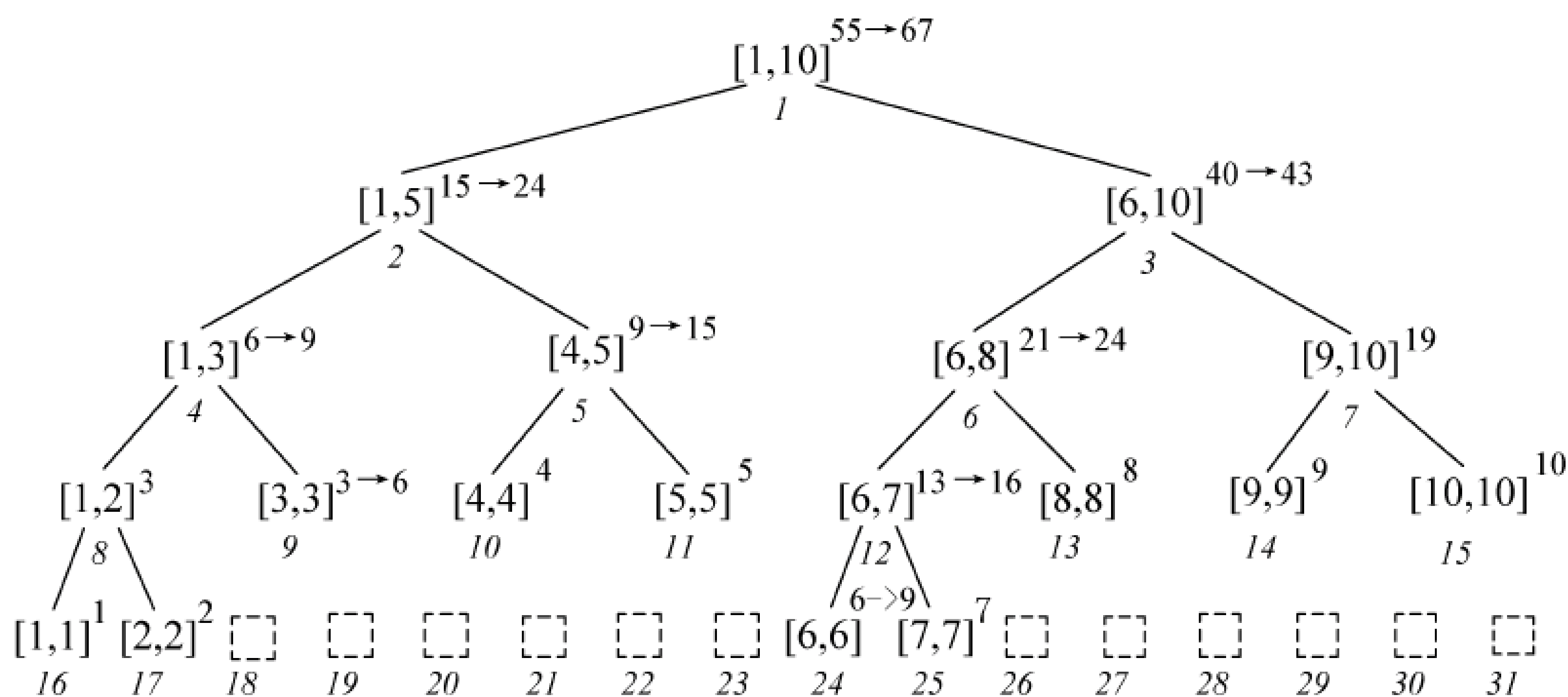


图 5.27　区间求和

(3) “Q a b”。同样可以利用 lazy 原理,当某个子区间包含在被查询的区间内时,直接返回这个子区间的区间和,不用继续深入。

下面是 poj 3468 的程序。build()函数建树,建树的结果见图 5.27; update()函数完成“C a b c”操作,query()函数完成“Q a b”操作。

sum[i]记录结点 i 的区间和,在图 5.27 中是结点右上角的数字。

add[i]是 tag,它记录结点 i 是否用到 lazy 原理,其值是“C a b c”中的 c; 如果做了多次 lazy,add[i]可以累加。一旦结点 i 在某次“C a b c”中被深入,破坏了 lazy,就把 add[i]归零,push_down()函数完成这一任务。

```
#include <stdio.h>
using namespace std;
const int MAXN = 1e5 + 10;
long long sum[MAXN << 2], add[MAXN << 2];        //4 倍空间
void push_up(int rt){                            //向上更新,通过当前结点 rt 把值递归到父结点
    sum[rt] = sum[rt << 1] + sum[rt << 1 | 1];
}
void push_down(int rt, int m){                   //更新 rt 的子结点
    if(add[rt]){
        add[rt << 1] += add[rt];
        add[rt << 1 | 1] += add[rt];
        sum[rt << 1] += (m - (m >> 1)) * add[rt];
        sum[rt << 1 | 1] += (m >> 1) * add[rt];
        add[rt] = 0;                             //取消本层标记
    }
}
#define lson l, mid, rt << 1
#define rson mid + 1, r, rt << 1 | 1
void build(int l, int r, int rt){                //用满二叉树建树
    add[rt] = 0;
    if(l == r){                                  //叶子结点,赋值
        scanf("%lld", &sum[rt]);
        return;
    }
    int mid = (l + r) >> 1;
    build(lson);
    build(rson);
    push_up(rt);                                 //向上更新区间和
}
void update(int a, int b, long long c, int l, int r, int rt){     //区间更新
    if(a <= l && b >= r){
        sum[rt] += (r - l + 1) * c;
        add[rt] += c;
        return;
    }
    push_down(rt, r - l + 1);                    //向下更新
    int mid = (l + r) >> 1;
    if(a <= mid) update(a, b, c, lson);          //分成两半,继续深入
    if(b > mid) update(a, b, c, rson);
    push_up(rt);                                 //向上更新
}
```

```
long long query(int a, int b, int l, int r, int rt){   //区间求和
    if(a <= l && b >= r) return sum[rt];               //满足 lazy,直接返回值
    push_down(rt, r - l + 1);                          //向下更新
    int mid = (l + r) >> 1;
    long long ans = 0;
    if(a <= mid) ans += query(a, b, lson);
    if(b > mid) ans += query(a, b, rson);
    return ans;
}
int main(void){
    int n, m;
    scanf("%d%d", &n, &m);
    build(1, n, 1);
    while(m--){
        char str[2];
        int a, b; long long c;
        scanf("%s", str);
        if(str[0] == 'C'){
            scanf("%d%d%lld", &a, &b, &c);
            update(a, b, c, 1, n, 1);
        }else{
            scanf("%d%d", &a, &b);
            printf("%lld\n", query(a, b, 1, n, 1));
        }
    }
}
```

5.3.5　线段树习题

简单题：hdu 1166/1394/1698/1754/2795；

poj 1195/2182/2299/2828/2352/2750/2886/2777/3264/3468。

中等题：hdu 1540/1823/4027/5869；

poj 2155/2528/2823/3225。

综合题：hdu 1255/1542/3642/3974/4578/4614/4718/5756/4441。

5.4　树状数组

树状数组(Binary Indexed Tree,BIT)是一种利用数的二进制特征进行检索的树状结构。树状数组是一种奇妙的数据结构,不仅非常高效,而且代码极其简洁。

1. 树状数组的概念

从下面这个例子引导出树状数组的概念。

长度为 n 的数列$\{a_1,a_2,\cdots,a_n\}$,进行以下操作。

(1) 修改元素 add(k,x)：把 a_k 加上 x。

(2) 求和 sum(x)：$x \leqslant n$,sum$=a_1+a_2+\cdots+a_x$。那么,区间和 $a_i+\cdots+a_j=$sum$(j)-$sum$(i-1)$。

这个程序很好写，用循环加或者前缀和，复杂度是 $O(n)$。然而，如果 n 很大，这样做的效率会非常低。读者可以用前面讲的线段树来实现高效的算法。其实有一种更好的数据结构，即树状数组，不仅效率和线段树一样高，只有 $O(\log_2 n)$，而且代码短得不可思议。先看一看代码：

```
#define lowbit(x)  ((x) & - (x))
void add(int x, int d) {                  //更新数组 tree[]。ax = ax + d,修改和 ax 有关的 tree[]
    while(x <= n) {
        tree[x] += d;
        x += lowbit(x);
    }
}
int sum(int x) {                          //求和：sum = a1 + a2 + … + ax
    int sum = 0;
    while(x > 0){
        sum += tree[x];
        x -= lowbit(x);
    }
    return sum;
}
```

add()和 sum()的复杂度都是 $O(\log_2 n)$。

上述代码的使用方法如下：

(1) 初始化，add()。先清空数组 tree[]，然后读取 $a_1, a_2, \cdots, a_n$，用 add()逐一处理这 n 个数，得到 tree[]数组。在程序中并不需要定义数组 a[]，因为它隐含在 tree[]中。

(2) 求和，sum()。计算 $\text{sum}=a_1+a_2+\cdots+a_x$，即执行 sum()。求和是基于数组 tree[]的。

(3) 如果需要修改元素，执行 add()，即修改数组 tree[]。

下面详细说明上述操作的原理。

2. lowbit()操作

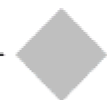

视频讲解

从代码中可以看出，其核心是一个神奇的 lowbit(x)操作。lowbit(x)= x & $-x$，功能是找到 **x 的二进制数的最后一个 1**。其原理是利用负数的补码表示，补码是原码取反加一。例如 $x=6=00000110_2$，$-x=x_{补}=11111010_2$，那么 lowbit(x)= x & $-x=10_2=2$。

1～9 的 lowbit()结果如表 5.1 所示。

表 5.1　1～9 的 lowbit()结果

x	1	2	3	4	5	6	7	8	9
x 的二进制	1	10	11	100	101	110	111	1000	1001
lowbit(x)	1	2	1	4	1	2	1	8	1
tree[x]数组	tree[1] $=a_1$	tree[2] $=a_1+a_2$	tree[3] $=a_3$	tree[4] $=a_1+a_2+a_3+a_4$	tree[5] $=a_5$	tree[6] $=a_5+a_6$	tree[7] $=a_7$	tree[8] $=a_1+a_2+\cdots+a_8$	tree[9] $=a_9$



lowbit(x)有什么用呢？从 lowbit(x)引出一个 tree[]数组，所有的计算都围绕 tree[]进行。

令 m=lowbit(x)，定义 tree[x]的值，是把 a_x 和它前面共 m 个数相加的结果，如表 5.1 所示。例如 lowbit(6)=2，tree[6]=a_5+a_6。

图 5.28 中的横线重新描述了这个关系，横线中的黑色表示 tree[x]，它等于横线上元素相加的和。

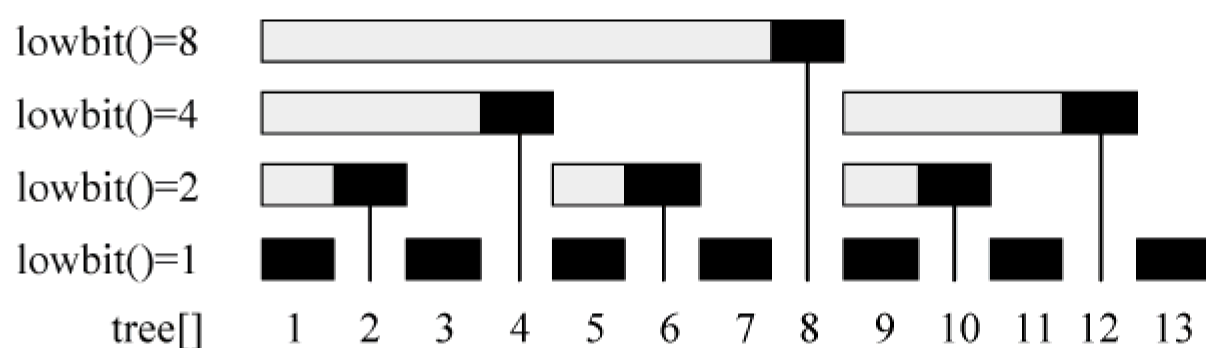


图 5.28 lowbit()计算

求和计算以及 tree[]数组的更新都可以通过 lowbit()完成。

1）求和计算 sum=$a_1+a_2+\cdots+a_x$

可以借助 tree[]数组求 sum，例如：

sum(8)=tree[8]

sum[7]=tree[7]+tree[6]+tree[4]

sum[9]=tree[9]+tree[8]

然而，如何得到上面的关系呢？

很容易观察到，在计算 sum 时，对 tree[]的查找可以通过 lowbit(x)实现。例如 sum[7]=tree[7]+tree[6]+tree[4]。

首先从 7 开始，加上 tree[7]；

然后 7−lowbit(7)=6，加上 tree[6]；

接着 6−lowbit(6)=4，加上 tree[4]；

最后 4−lowbit(4)=0，结束。

编程细节见前面的求和函数 sum()，复杂度是 $O(\log_2 n)$。

2）tree[]数组的更新

更改 a_x，那么和它相关的 tree[]都会变化。例如改变 a_3，那么 tree[3]、tree[4]、tree[8]等都会改变。同样，这个计算也利用了 lowbit(x)。

首先更改 tree[3]；

然后 3+lowbit(3)=4，更改 tree[4]；

接着 4+lowbit(4)=8，更改 tree[8]；

继续，直到最后的 tree[n]。

编程细节见函数 add()，复杂度也是 $O(\log_2 n)$。add()函数也用于 tree[]的初始化过程：tree[]初始化为 0，然后用 add()逐一处理 $a_1,a_2,\cdots,a_n$。

3. 例题

这里仍然以 poj 2182 为例，用树状数组实现。该题用树状数组更容易理解。

其中的关键点如下：

(1) 在 n 个位置上,每个位置有一头牛,即 $a_1=a_2=\cdots=a_n=1$。不过,在程序中并不需要直接定义和使用数组 a[]。

(2) tree[]数组的初始化。这个题目比较特殊,不需要用 add()初始化,因为 lowbit(i)就是 tree[i]。

(3) 程序所做的,就是对每个 pre[i]+1,用 findpos()找出 sum(x)=pre[i]+1 所对应的 x,就是第 x 头牛。在找到第 x 头牛之后,令 $a_x=0$,方法是用 add()更新数组 tree[],即执行 add(x, -1)。

下面的程序完全套用了上面提到的树状数组的模板。

poj 2182 "树状数组"

```
#include <stdio.h>
#include <string.h>
const int Max = 10000;
int tree[Max], pre[Max], ans[Max];
int n;
#define lowbit(x)  ((x) & - (x))
void add(int x, int d){
    while(x <= n) {
        tree[x] += d;
          x += lowbit(x);
    }
}
int sum(int x){
    int sum = 0;
    while(x > 0) {
        sum += tree[x];
          x -= lowbit(x);
    }
    return sum;
}
int findpos(int x){                    //寻找 sum(x) = pre[i] + 1 所对应的 x,就是第 x 头牛
    int l = 1, r = n;
    while(l < r) {
        int mid = (l + r) >> 1;
        if(sum(mid) < x)
            l = mid + 1;
        else
            r = mid;
    }
    return l;
}
int main(){
    scanf("%d",&n);
    pre[1] = 0;
    for(int i = 2; i <= n; i++)
        scanf("%d",&pre[i]);
    for(int i = 1; i <= n; i++)          //初始化 tree[]数组
          //注意这个题目比较特殊,不需要用 add()初始化,因为 lowbit(i)就是 tree[i]
```



```
        tree[i] = lowbit(i);
    for(int i = n; i > 0; i-- )  {
        int x = findpos(pre[i] + 1);
        add(x, - 1);                    //更新 tree[]数组
        ans[i] = x;
    }
    for(int i = 1; i <= n; i++)
        printf(" %d\n", ans[i]);
    return 0;
}
```

4. 线段树和树状数组的对比

两者的复杂度同级，但是树状数组的常数明显优于线段树，编程复杂度也远远小于线段树。

线段树的适用范围大于树状数组，凡是可以使用树状数组解决的问题，使用线段树一定可以解决。树状数组的优点是编程非常简洁，使用 lowbit()可以在很短的几步操作中完成核心操作，代码效率远远高于线段树。

【习题】

简单题：poj 2299/2352/1195/2481/2029。
中等题：poj 2155/3321/1990；
　　　　hdu 3015/2430/2852。
难题：poj 2464，uva 11610。

5.5 小　　结

本章介绍了几个竞赛中常用的数据结构，限于篇幅，还有一些常用的数据结构没讲，例如堆、Hash、动态树 LCT 等。关于字符串的数据结构，在第 9 章中讲解；关于图的数据结构，在第 10 章中讲解。

高级数据结构是算法竞赛中比较难的内容，不仅本身的概念难以掌握，而且在具体的题目中需要根据情况灵活修改，以至于逻辑复杂、代码冗长。

第 6 章　基础算法思想

✍贪心法
✍Huffman 编码
✍分治法
✍归并排序
✍快速排序
✍减治法

在竞赛中，队员拿到一个题目后很快就能知道这个题的考点是什么，例如图论、几何、数学、模拟、高级数据结构等。有时候老队员还会说："这一题的思路是动态规划……"

这里提到的动态规划并不是一个具体的算法，而是一种算法思想，或者是解题策略。类似地，把算法思想分成一些大类[①]，即暴力法、分治法、减治法、贪心法、动态规划。

本章将详细介绍贪心法、分治法、减治法。暴力法已经在"第 4 章 搜索技术"中介绍，动态规划将在"第 7 章 动态规划"中详细展开。

对于算法竞赛初学者来说，从只会按自然理解和逻辑做题，到能使用算法思想分析和设计，建立起基本的计算思维意识，是成为高级编程者的重要一步。

6.1 贪　心　法

6.1.1　基本概念

贪心(Greedy)是最容易理解的算法思想：把整个问题分解成多个步骤，在每个步骤都选取当前步骤的最优方案，直到所有步骤结束；在每一步都不考虑对后续步骤的影响，在后续步骤中也不再回头改变前面的选择。简单地说，其思想就是"走一步看一步""目光短浅"。

贪心法看起来似乎不靠谱，因为局部最优的组合不一定是全局最优的。那么，是否有一些规则使得局部最优能达到全局最优？本节将通过一些例子来详细说明这个问题。

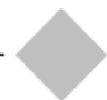
贪心法有广泛的应用。例如图论中的最小生成树算法、单源最短路径算法 Dijkstra 是贪心思想的典型应用。关于这部分内容，请阅读"第 10 章 图论"。

下面先用硬币问题的例子引出贪心法的应用规则。

最少硬币问题：某人带着 3 种面值的硬币去购物，有 1 元、2 元、5 元的，硬币数量不限；他需要支付 M 元，问怎么支付才能使硬币数量最少？

---

① 讲解算法的经典教材《算法设计与分析基础》就是按这个分类展开的，由 Anany Levitin 著、潘彦译。另一本经典教材《算法导论》由 Thomas H. Cormen 等著、潘金贵等译，主要是按知识点内容来展开。



根据生活常识，第一步应该先拿出面值最大的 5 元硬币，第二步拿出面值第 2 大的 2 元硬币，最后才拿出面值最小的 1 元硬币。在这个解决方案中，硬币数量总数是最少的。

程序如下：

```
#include<bits/stdc++.h>
using namespace std;
const int NUM = 3;
const int Value[NUM] = {1,2,5};
int main(){
    int i, money;
    int ans[NUM] = {0};                //记录每种硬币的数量
    cin >> money;                      //输入钱数
    for(i= NUM-1; i>=0; i--){          //求每种硬币的数量
        ans[i] = money/Value[i];
        money = money - ans[i] * Value[i];
    }
    for(i= NUM-1; i>=0; i--)
        cout << Value[i] << "元硬币数:" << ans[i] << endl;
    return 0;
}
```

在上面的例子中，虽然每一步选硬币的操作并没有从整体最优来考虑，只在当前步骤选取了局部最优，但结果是全局最优的。然而，局部最优并不总是能导致全局最优。比如这个最少硬币问题，用贪心法一定能得到最优解吗？

在最少硬币问题中，如果稍微改一下参数，就不一定能得到最优解，甚至在有解的情况下也无法算出答案。

（1）不能得到最优解的情形。例如，硬币面值比较奇怪，是 1、2、4、5、6 元，支付 9 元，如果用贪心法，答案是 6＋2＋1，需要 3 个硬币，而最优的 5＋4 只需要两个硬币。

（2）算不出答案的情形。例如，如果有面值 1 元的硬币，能保证用贪心法得到一个解，如果没有 1 元硬币，常常得不到解。用面值 2、3、5 元的硬币，支付 9 元，用贪心法无法得到解，但解是存在的，即 9＝5＋2＋2。

所以，在最少硬币问题中是否能使用贪心法跟硬币的面值有关。如果是 1、2、5 这样的面值，贪心法是有效的，而对于 1、2、4、5、6 或者 2、3、5 这样的面值，贪心法是无效的①。对任意面值的硬币问题，需要用动态规划求最优解，在下一章讲解动态规划时会提到。

虽然贪心法不一定能得到最优解，但是它思路简单、编程容易。因此，如果一个问题确定用贪心法能得到最优解，那么应该使用它。

那么，如何判断一个题目能用贪心法？用贪心法求解的问题需要满足以下特征：

（1）最优子结构性质。当一个问题的最优解包含其子问题的最优解时，称此问题具有最优子结构性质，也称此问题满足最优性原理。也就是说，从局部最优能扩展到全局最优。

① 一个简单的判断标准是，面值符合 $c_j > \sum_{i=1}^{i<j-1} c_i$ 的硬币，即任一面值的硬币，大于比它小的所有硬币的面值之和，可以用贪心法。例如以 2 的倍数递增的 1、2、4、8 等，这样的面值就符合条件。

（2）贪心选择性质。问题的整体最优解可以通过一系列局部最优的选择来得到。

贪心算法没有固定的算法框架，关键是如何选择贪心策略。贪心策略必须具备无后效性，即某个状态以后的过程不会影响以前的状态，只与当前状态有关。

另外，对于某些难解问题，例如旅行商问题，很难得到最优解，但是此时用贪心法常常能得到不错的近似解。如果不一定非要求得最优解，那么贪心的结果也是很不错的方案。

6.1.2 常见问题



视频讲解

1. 活动安排问题

活动安排问题又称为区间调度问题，原型见 hdu 2037 题。

> **hdu 2037 "今年暑假不 AC"**
>
> 有很多电视节目，给出它们的起止时间，有的节目时间冲突，问能完整看完的电视节目最多有多少？

解题的关键在于选择什么贪心策略才能安排尽量多的活动。由于活动有开始时间和结束时间，考虑下面 3 种贪心策略：

（1）最早开始时间。

（2）最早结束时间。

（3）用时最少。

经过分析发现，第 1 种策略是错误的，因为如果一个活动迟迟不终止，后面的活动就无法开始。第 2 种策略是合理的，一个尽快终止的活动可以容纳更多的后续活动。第 3 种策略也是错误的。

对最早结束时间进行贪心，算法步骤如下：

（1）把 n 个活动按结束时间排序。

（2）选择第 1 个结束的活动，并删除（或跳过）与它时间相冲突的活动。

（3）重复步骤（2），直到活动为空。每次选择剩下的活动中最早结束的那个活动，并删除与它时间冲突的活动。

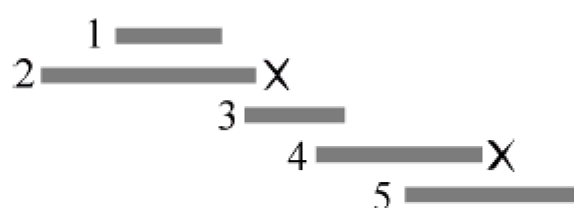


图 6.1　活动安排

下面的图 6.1 是例子，最优活动是 1、3、5，活动 2 和活动 4 与其他节目有冲突。

上述贪心算法是否能保证得到全局最优解？

（1）它符合最优子结构性质。选中的第 1 个活动，它一定在某个最优解中；同理，选中的第 2 个活动、第 3 个活动等也都在这个最优解中。

（2）它符合贪心选择性质。算法的每一步都使用了相同的贪心策略。

hdu 2037 部分代码

```
struct node {
    int start, end;                    //定义活动的起止时间
} record[MAXN];
bool cmp(const node& a, const node& b){return a.end < b.end; }
```

```
for(int i = 0; i < n; i++)                    //输入 n 个活动
    cin >> record[i].start >> record[i].end;
sort(record, record + n, cmp);                //按结束时间排序
int count = 0;
int lastend = -1;
for(int i = 0; i < n; i++) {                  //贪心算法
    if(record[i].start >= lastend){           //后一个起始时间大于等于前一个终止时间
        count++;
        lastend = record[i].end;              //记录前一个活动的终止时间
    }
}
cout << count << endl;                        //输出活动个数
```

2. 区间覆盖问题

给定一个长度为 n 的区间，再给出 m 条线段的左端点（起点）和右端点（终点），问最少用多少条线段可以将整个区间完全覆盖？

贪心思路是尽量找出更长的线段。其解题步骤如下：

（1）把每个线段按照左端点递增排序。

（2）设已经覆盖的区间是$[L,R]$，在剩下的线段中找所有左端点小于等于 R 且右端点最大的线段，把这个线段加入到已覆盖区间里，并更新已覆盖区间的$[L,R]$值。

（3）重复步骤(2)，直到区间全部覆盖。

在图 6.2 中，所有线段已按左端点进行排序。首先选中线段 1，然后在 2 和 3 中选中更长的 3。4 和 5 由于不合要求，被跳过。最后的最优解是 1、3。

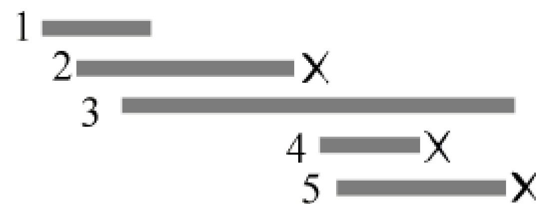


图 6.2　区间覆盖

3. 最优装载问题

原型见 hdu 2570 题。

> **hdu 2570 “迷瘴”**
>
> 有 n 种药水，体积都是 V，浓度不同，把它们混合起来，得到浓度不大于 $w\%$ 的药水，问怎么混合才能得到最大体积的药水？注意一种药水要么全用，要么都不用，不能只取一部分。

题目要求配置浓度不大于 $w\%$ 的药水，那么贪心的思路就是尽量找浓度小的药水。先对药水按浓度从小到大排序，药水的浓度不大于 $w\%$ 就加入，如果药水的浓度大于 $w\%$，计算混合后的总浓度，不大于 $w\%$ 就加入，否则结束判断。

4. 多机调度问题

设有 n 个独立的作业，由 m 台相同的计算机进行加工。作业 i 的处理时间为 t_i，每个作业可在任何一台计算机上加工处理，但不能间断、拆分。要求给出一种作业调度方案，在尽可能短的时间内，由 m 台计算机加工处理完成这 n 个作业。

求解多机调度问题的贪心策略是最长处理时间的作业优先，即把处理时间最长的作业分配给最先空闲的计算机。让处理时间长的作业得到优先处理，从而在整体上获得尽可能

短的处理时间。

(1) 如果 $n \leqslant m$，需要的时间就是 n 个作业当中最长的处理时间 t。

(2) 如果 $n > m$，首先将 n 个作业按处理时间从大到小排序，然后按顺序把作业分配给空闲的计算机。

6.1.3 Huffman 编码

Huffman 编码是贪心思想的典型应用，是一个很有用的、很著名的算法。Huffman 编码是"前缀"最优编码。

首先了解什么是编码。

把一段字符串存储在计算机中，这段字符串包含很多字符，每种字符出现的次数不一样，有的频次高，有的频次低。因为数据在计算机中都是用二进制码来表示的，所以需要把每个字符编码成一个二进制数。

最简单的编码方法是把每个字符都用相同长度的二进制数来表示。例如给出一段字符串，它只包含 A、B、C、D、E 这 5 种字符，编码方案如表 6.1 所示。

表 6.1　简单编码方案

字　　符	A	B	C	D	E
频　　次	3	9	6	15	19
编　　码	000	001	010	011	100

每个字符用 3 位二进制数表示，存储的总长度是 $3\times(3+9+6+15+19)=156$。

这种编码方法简单、实用，但是不节省空间。由于每个字符出现的频次不同，可以想到用**变长编码**：出现次数多的字符用短码表示，出现少的用长码表示，例如表 6.2。

表 6.2　变长编码方案

字　　符	A	B	C	D	E
频　　次	3	9	6	15	19
编　　码	1100	111	1101	10	0

存储的总长度是 $3\times4+9\times3+6\times4+15\times2+19\times1=112$。

第 2 种方法相当于第 1 种方法进行了压缩，压缩比是 $156/112=1.39$。

当然，编码算法的基本要求是编码后得到的二进制串能唯一地进行解码还原。上面第 1 种方法是正确的，每 3 位二进制数对应一个字符。第 2 种方法也是正确的，例如"11001111001101"，解码后唯一得到"ABDEC"。

如果胡乱设定编码方案，很可能是错误的，例如表 6.3。

表 6.3　错误编码方案

字　　符	A	B	C	D	E
频　　次	3	9	6	15	19
编　　码	100	10	11	1	0

看起来似乎每个字符都有不同的编码，编码后的总长度也更短，只有 3×3+9×2+6×2+15×1+19×1=73。但是编码无法解码还原，例如"100"，是"A"、"BE"还是"DEE"呢？

错误的原因是，某个编码是另一个编码的**前缀**(prefix)，即这两个编码有包含关系，导致了混淆。

那么有没有比第 2 种编码方法更好的方法？这引出了一个字符串存储的常见问题：给定一个字符串，如何编码，能使编码后的总长度最小？即如何得到一个最优解？

作为后续讲解的预习，读者可以验证：第 2 种编码方法已经达到了最优，编码后的总长度 112 就是能得到的最小长度。

下面介绍 Huffman 编码。Huffman 编码是前缀编码算法中的最优算法。

首先考虑如何进行编码？由于编码是二进制，容易想到用二叉树来构造编码。

例如上面第 2 种编码方案，其二叉树如图 6.3 所示。

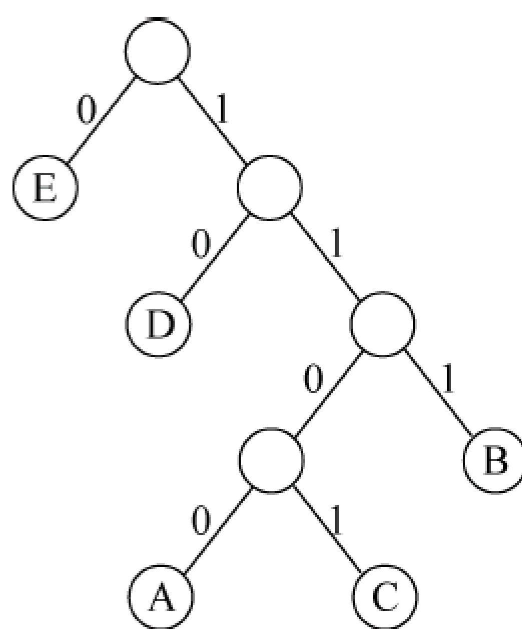


图 6.3　用二叉树实现前缀编码

在每个二叉树的分支，左边是 0，右边是 1。二叉树末端的叶子是编码，把编码放在叶子上，可以保证符合前缀不包含的要求。出现频次最高的字符 E，在最靠近根的位置，编码最短；出现频次最低的字符 A，在二叉树最深处，编码最长。

这棵编码二叉树是如何构造的？是最优的吗？

Huffman 编码是利用贪心思想构造二叉编码树的算法。

首先对所有字符按出现频次排序，如表 6.4 所示。

表 6.4　对字符按出现频次排序

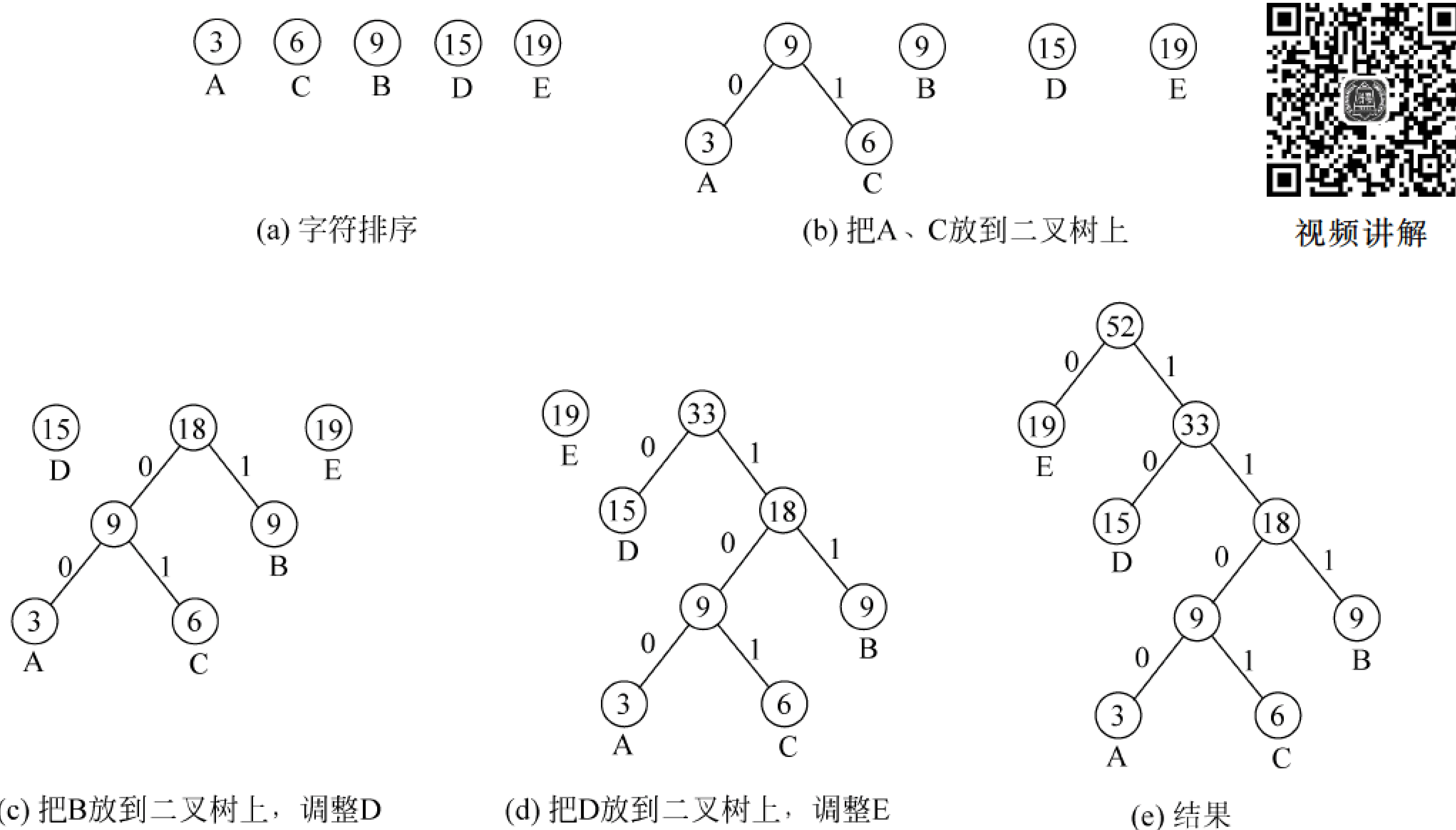
字　　符	A	C	B	D	E
频　　次	3	6	9	15	19

然后从出现频次最少的字符开始，用贪心思想安排在二叉树上。其步骤如图 6.4 所示。每个结点圆圈内的数字是这个子树下字符出现的频次之和。

贪心的过程是按出现频次从底层往顶层生成二叉树。注意，每一步都要按频次重新排序，例如图 6.4(c)和(d)中调整了 D 和 E 的顺序。这个过程可以保证出现频次少的字符被放在树的底层，编码更长；出现多的字符被放在上层，编码更短。

可以证明，Huffman 算法符合贪心法的“最优子结构性质”和“贪心选择性质”[①]。编码的结果是最优的。

① 证明见《算法导论》，Thomas H. Cormen 等著，潘金贵等译，机械工业出版社，234 页，“赫夫曼算法的正确性”。




视频讲解

图 6.4　Huffman 编码算法的步骤

下面给出一个例题。

> **poj 1521 "Entropy"**
>
> 输入一个字符串，分别用普通 ASCII 编码（每个字符 8bit）和 Huffman 编码，输出编码后的长度，并输出压缩比。
>
> 输入样例：
>
> AAAAABCD
>
> 输出样例：
>
> 64 13 4.9

这一题正常的解题过程是首先统计字符出现的频次，然后用 Huffman 算法编码，最后计算编码后的总长度。不过，由于只需要输出编码的总长度，而不要求输出每个字符的编码，所以可以跳过编码过程，利用图 6.4 描述的 Huffman 编码思想（圆圈内的数字是出现频次），直接计算编码的总长度。

下面的代码使用了 STL 的优先队列，在每个贪心步骤，从优先队列中提取频次最低的两个字符。

poj 1521 部分代码

```
string s;
priority_queue < int, vector < int >, greater < int >> Q;
                                        //优先队列,最小的在队首
while(getline(cin, s) && s != "END"){   //输入字符串
    int t = 1;
    sort(s.begin(), s.end());
    for(int i = 1;i < s.length();i++){   //统计字符出现的频次,并放进优先队列
```

```
            if(s[i] != s[i-1]){
                Q.push(t);
                t = 1;
            }
            else t++;
        }
        Q.push(t);
        int ans = 0;
        while(Q.size() > 1){
            int a = Q.top(); Q.pop();  //提取队列中最小的两个
            int b = Q.top(); Q.pop();
            Q.push(a + b);
            ans += a + b;              //直接计算编码的总长度,请思考为什么
        }
        Q.pop();
    }
    //ans 就是编码后的总长度
```

6.1.4　模拟退火

模拟退火算法基于这样一个物理原理：一个高温物体降温到常温，温度越高时降温的概率越大（降温更快），温度越低时降温的概率越小（降温更慢）。模拟退火算法利用这样一种思想进行搜索，即进行多次降温（迭代），直到获得一个可行解。

在迭代过程中，模拟退火算法随机选择下一个状态，有两种可能：①新状态比原状态更优，那么接受这个新状态；②新状态更差，那么以一定的概率接受该状态，不过这个概率应该随着时间的推移逐渐降低。

模拟退火算法是贪心思想和概率的结合，常用"爬山"问题来介绍贪心有关的算法，在图 6.5 中，A 是局部最高点，B 是全局最高点。普通的贪心算法，如果当前状态在 A 附近，会一直爬山，最后停滞在局部最高点 A，而无法到达 B。模拟退火算法能跳出 A，得到 B。因为它不仅往上爬山，而且以一定的概率接受比当前点更低的点，使程序有机会摆脱局部最优到达全局最优。这个概率会随时间不断减小，从而最后能限制在最优解附近。

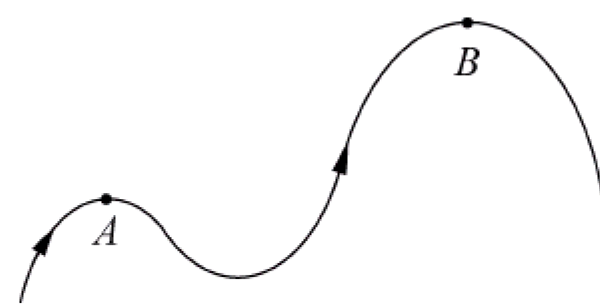


图 6.5　模拟退火与贪心

模拟退火算法的主要步骤如下：

（1）设置一个初始的温度 T。

（2）温度下降，状态转移。从当前温度按降温系数下降到下一个温度，在新的温度计算当前状态。

（3）如果温度降到设定的温度下界，程序停止。

伪代码如下：

```
eps = 1e-8;            //终止温度,接近 0,用于控制精度
T = 100;               //初始温度,应该是高温,以 100℃ 为例
delta = 0.98;          //降温系数,控制退火的快慢,小于 1,以 0.98 为例
g(x);                  //状态 x 时的评价函数,例如物理意义上的能量
```

```
now, next;                          //当前状态和新状态
while(T > eps){                     //如果温度未降到 eps
        g(next), g(now);            //计算能量
        dE = g(next) - g(now);      //能量差
        if(dE >= 0)                 //新状态更优,接受新状态
            now = next;
        else if(exp(dE/T)> rand())  //如果新状态更差,在一定概率下接受它,e^(dE/T)
            now = next;
        T *= delta;                 //降温,模拟退火过程
}
```

模拟退火在算法竞赛中的典型应用有函数最值问题、TSP 旅行商问题、最小圆覆盖、最小球覆盖等。在本书第 11.2.2 节中给出了用模拟退火求解最小圆覆盖的例子。下面的例子是求函数最值。

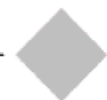
hdu 2899 “Strange function”

函数 $F(x)=6x^7+8x^6+7x^3+5x^2-yx$,其中 x 的范围是 $0\leqslant x\leqslant 100$。

输入 y 值,输出 $F(x)$的最小值。

用模拟退火求函数最值是最合适的。下面是代码:

```
#include <bits/stdc++.h>
using namespace std;
const double eps = 1e-8;                          //终止温度
double y;
double func(double x){                            //计算函数值
    return 6 * pow(x,7.0) + 8 * pow(x,6.0) + 7 * pow(x,3.0) + 5 * pow(x,2.0) - y * x;
}
double solve(){
    double T = 100;                               //初始温度
    double delta = 0.98;                          //降温系数
    double x = 50.0;                              //x 的初始值
    double now = func(x);                         //计算初始函数值
    double ans = now;                             //返回值
    while(T > eps){                               //eps 是终止温度
        int f[2] = {1, -1};
        double newx = x + f[rand() % 2] * T;      //按概率改变 x,随 T 的降温而减少
        if(newx >= 0 && newx <= 100){
            double next = func(newx);
            ans = min(ans,next);
            if(now - next > eps){x = newx; now = next;}  //更新 x
        }
        T *= delta;
    }
    return ans;
}
int main(){
    int cas; scanf("%d",&cas);
```



```
    while(cas--){
        scanf("%lf",&y);
        printf("%.4f\n",solve());
    }
}
```

模拟退火算法用起来非常简单、方便，不过也有缺点。它得到的是一个可行解，而不是精确解。例如上面的例题，计算到4位小数点的精度就停止，实际上是一个可行解，所以算法的效率和要求的精度有关。在一般情况下，模拟退火算法的复杂度会比其他精确算法差。用户在应用时需要仔细选择初始温度 T、降温系数 delta、终止温度 eps 等。

6.1.5　习题

hdu 1789 "Doing Homework again"，活动安排问题。

hdu 1050 "Moving Tables"，空间问题，模型和活动安排问题一样。

hdu 2546 "饭卡"，普通背包问题。

hdu 3348 "coins"，钱币问题。

hdu 4864 "task"，不错的题。

poj 1328 "Radar Installation"，几何问题，建模为活动安排问题。

poj 1089 "Intervals"，区间覆盖问题，给定很多线段，合并线段，使得合并后间隔最小。

6.2　分　治　法

分治法是广为人知的算法思想，很容易理解。人们在遇到一个难以直接解决的大问题时，自然会想到把它划分成一些规模较小的子问题，各个击破，"分而治之(Divide and Conquer)"。

在软件开发项目的详细设计阶段，常常会开一个"头脑风暴"会议，把整个项目分解成相对独立的子问题，其思想符合分治法。

分治算法的具体操作是把原问题分成 k 个较小规模的子问题，对这 k 个子问题分别求解。如果子问题不够小，那么把每个子问题再划分为规模更小的子问题。这样一直分解下去，直到问题足够小，很容易求出这些小问题的解为止。

能用分治法的题目需要符合以下两个特征。

(1) 平衡子问题：子问题的规模大致相同，能把问题划分成大小差不多相等的 k 个子问题，最好 $k=2$，即分成两个规模相等的子问题。子问题规模相等的处理效率比子问题规模不等的处理效率要高。

(2) 独立子问题：子问题之间相互独立。这是区别于动态规划算法的根本特征，在动态规划算法中，子问题是相互联系的，而不是相互独立的。

特别需要说明的是，分治法不仅能够让问题变得更容易理解和解决，而且能大大优化算法的复杂度，在一般情况下能把 $O(n)$ 的复杂度优化到 $O(\log_2 n)$。这是因为，**局部的优化有利于全局；一个子问题的解决，其影响力扩大了 k 倍，即扩大到了全局**。

举一个简单的例子：在一个有序的数列中查找一个数。简单的办法是从头找到尾，复杂度是 $O(n)$。如果用分治法，即“折半查找”，则最多只需要 $\log_2 n$ 次就能找到。

分治法是一种“并行”算法。由于子问题是相互独立的，因此可以把子问题分给不同的计算机，分开单独解决。

分治法如何编程？分治法的思想几乎就是递归的过程，用递归程序实现分治法是很自然的。

在用分治法建立模型时，解题步骤分为以下 3 步。

(1) 分解(Divide)：把问题分解成独立的子问题。

(2) 解决(Conquer)：递归解决子问题。

(3) 合并(Combine)：把子问题的结果合并成原问题的解。

分治法的经典应用有汉诺塔、快速排序、归并排序等。

6.2.1 归并排序

归并排序和快速排序都是非常精美的算法，学习它们，对于理解分治法思想、提高算法思维能力十分有帮助。在学习归并排序和快速排序之前，请读者先学习交换排序、选择排序、冒泡排序等暴力的排序方法[①]。

在介绍归并排序和快速排序之前先思考一个问题：如何用分治思想设计排序算法？

根据分治法的分解、解决、合并三步骤，具体思路如下：

(1) 分解。把原来无序的数列分成两部分，对每个部分，再继续分解成更小的两部分……在归并排序中，只是简单地把数列分成两半。在快速排序中，是把序列分成左、右两部分，左部分的元素都小于右部分的元素。分解操作是快速排序的核心操作。

(2) 解决。分解到最后不能再分解，排序。

(3) 合并。把每次分开的两个部分合并到一起。归并排序的核心操作是合并，其过程类似于交换排序。快速排序并不需要合并操作，因为在分解过程中左、右部分已经是有序的。

本节先讲解归并排序，然后讲解归并排序的典型应用——“逆序对”问题。

1. 归并排序示例

下面的例子给出了归并排序的操作步骤。初始数列经过 3 趟归并之后得到一个从小到大的有序数列，如图 6.6 所示。请读者根据这个例子分析它是如何实现分治法的分解、解决、合并 3 个步骤的。

分析该图，归并排序的主要操作如下：

(1) 分解。把初始序列分成长度相同的左、右两个子序列，然后把每个子序列再分成更小的两个子序列，直到子序列只包含 1 个数。这个过程用递归实现，图 6.6 中的第 1 行是初始序列，每个数是一个子序列，可以看成递归到达的最底层。

(2) 求解子问题，对子序列排序。最底层的子序列只包含 1 个数，其实不用排序。

---

① 算法竞赛中的排序，最多只处理千万级的数据量，即可以一次在内存中处理。工程上可能需要对大数据排序，例如 1TB 的数据，数据量太大，单个的 CPU 一次只能处理一小部分，所以不能简单地用某个排序算法。在找工作面试时，常常出现这种大数据排序的题目，读者可以学习有关的知识。

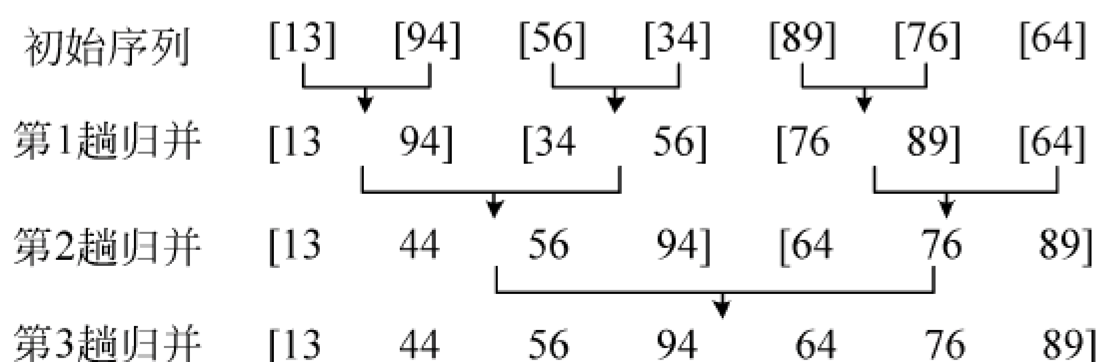


图 6.6　归并排序

(3) 合并。归并两个有序的子序列，这是归并排序的主要操作，过程如图 6.7 所示。例如在图 6.7(a)中，i 和 j 分别指向子序列{13,94,99}和{34,56}的第 1 个数，进行第 1 次比较，发现 $a[i]<a[j]$，把 $a[i]$放到临时空间 $b[]$中。总共经过 4 次比较，得到了 $b[]=\{13,34,56,94,99\}$。

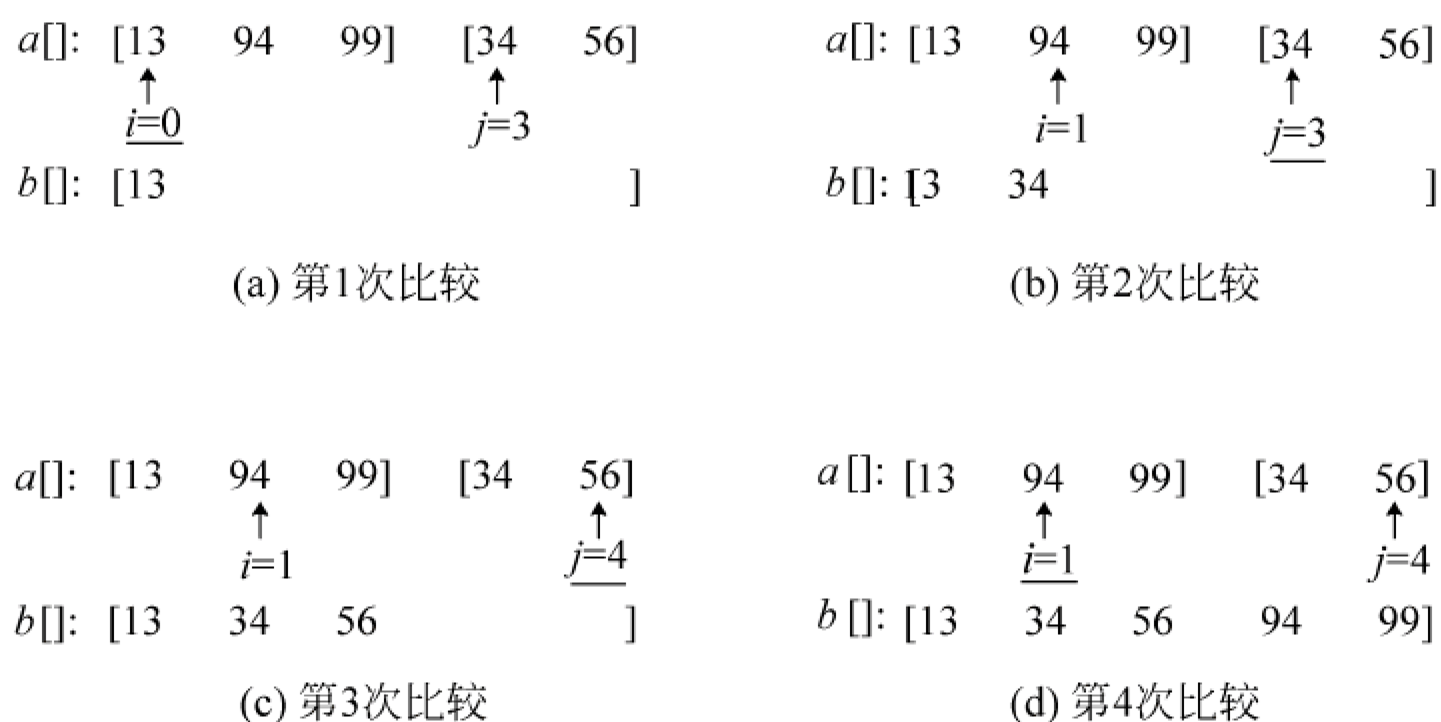


图 6.7　归并排序的一次合并

在暴力排序算法中，有一种算法是交换排序，归并排序可以看成是交换排序的升级版。

交换排序的步骤如下：

(1) 第 1 轮，检查第 1 个数 a_1。把序列中后面所有的数一个一个跟它比较，如果发现有一个比 a_1 小，就交换。第 1 轮结束后，最小的数就排在了第 1 个位置。

(2) 第 2 轮，检查第 2 个数。第 2 轮结束后，第 2 小的数排在了第 2 个位置。

(3) 继续上述过程，直到检查完最后一个数。

交换排序的复杂度是 $O(n^2)$。

在归并排序中，一次合并的操作和交换排序很相似，只是合并的操作是基于两个有序的子序列，效率更高。

下面分析归并排序的**计算复杂度**。对 n 个数进行归并排序：①需要 $\log_2 n$ 趟归并；②在每一趟归并中有很多次合并操作，一共需要 $O(n)$ 次比较。所以计算复杂度是 $O(n\log_2 n)$。

空间复杂度：由于需要一个临时的 $b[]$存储结果，所以空间复杂度是 $O(n)$。

读者从归并排序的例子可以体会到，对于整体上 $O(n)$ 复杂度的问题，通过分治可以减少为 $O(\log_2 n)$复杂度的问题。

2. 逆序对问题

排序是竞赛中的常用功能，一般直接使用 STL 的 sort()函数，并不需要自己再写一个

排序的程序。不过也有一些特殊的问题，需要写出程序，并在程序内部做一些处理，例如逆序对问题。

hdu 4911 "Inversion"

输入一个序列$\{a_1,a_2,\cdots,a_n\}$，交换任意两个相邻元素，不超过k次。在交换之后，问最少的逆序对有多少个?

序列中的一个**逆序对**是指存在两个数a_i和a_j，有$a_i>a_j$且$1\leqslant i<j\leqslant n$。也就是说，大的数排在小的数前面。

输入：第1行是n和k，$1\leqslant n\leqslant 10^5$，$0\leqslant k\leqslant 10^9$；第2行包括$n$个整数$\{a_1,a_2,a_3,\cdots,a_n\}$，$0\leqslant a_i\leqslant 10^9$。

输出：最少的逆序对数量。

输入样例：

3 1

2 2 1

输出样例：

1

当$k=0$时，就是求原始序列中有多少个逆序对。

求$k=0$时的逆序对，用暴力法很容易：先检查第1个数a_1，把后面的所有数跟它比较，如果发现有一个比a_1小，就是一个逆序对；再检查第2个数，第3个数……直到最后一个数。其复杂度是$O(n^2)$。本题中n最大是10^5，所以暴力法会TLE。

考察暴力法的过程，会发现和交换排序很像。那么自然可以想到，能否用交换排序的升级版——归并排序来处理逆序对问题?

观察图6.7所示的一次合并过程发现，可以利用这个过程记录逆序对。观察到以下现象：

(1) 在子序列内部，元素都是有序的，不存在逆序对；逆序对只存在于不同的子序列之间。

(2) 在合并两个子序列时，如果前一个子序列的元素比后面子序列的元素小，那么不产生逆序对，如图6.7(a)所示；如果前一个子序列的元素比后面子序列的元素大，就会产生逆序对，如图6.7(b)所示。不过，在一次合并中，产生的逆序对不止一个，例如在图6.7(b)中把34放到b[]中时，它与94、99产生了两个逆序对。在下面的程序中，相关代码是"cnt+=mid−i+1;"。

根据以上观察，只要在归并排序过程中记录逆序对就行了。

以上解决了$k=0$时原始序列中有多少个逆序对的问题，现在考虑，当$k\neq 0$时(即把序列中任意两个相邻数交换**不超过**k次)逆序对最少有多少?注意，不超过k次的意思是可以少于k次，而不是一定要k次。

在所有相邻数中，只有交换那些逆序的才会影响逆序对的数量。设原始序列有cnt个逆序对，讨论以下两种情况：

(1) 如果cnt$\leqslant k$，总逆序数量不够交换k次。所以进行k次交换之后，最少的逆序对数



量为 0。

(2) 如果 cnt>k,让 k 次交换都发生在逆序的相邻数上,那么剩余的逆序对是 cnt−k。

求逆序对的程序几乎可以完全套用归并排序的模板,差不多就是归并排序的裸题。在下面的程序中,Mergesort()和 Merge()是归并排序。与纯归并排序的程序相比,它只多了一句"cnt+=mid−i+1;"。

hdu 4911 归并排序(求逆序对)

```
#include <bits/stdc++.h>
const int MAXN = 100005;
typedef  long  long  ll;
ll a[MAXN], b[MAXN], cnt;
void Merge(ll l, ll mid, ll r){
    ll i = l, j = mid + 1, t = 0;
    while(i <= mid && j <= r){
        if(a[i] > a[j]){
            b[t++] = a[j++];
            cnt += mid - i + 1;                   //记录逆序对数量
        }
        else b[t++] = a[i++];
    }
  //一个子序列中的数都处理完了,另一个还没有,把剩下的直接复制过来
    while(i <= mid)   b[t++] = a[i++];
    while(j <= r)     b[t++] = a[j++];
    for(i = 0; i < t; i++)  a[l + i] = b[i];      //把排好序的 b[]复制回 a[]
}
void Mergesort(ll l, ll r){
    if(l < r){
        ll  mid = (l + r)/2;                       //平分成两个子序列
        Mergesort(l, mid);
        Mergesort(mid + 1, r);
        Merge(l, mid, r);                          //合并
    }
}
int main(){
    ll n,k;
    while(scanf("%lld%lld", &n, &k) != EOF){
        cnt = 0;
        for(ll i = 0;i < n;i++)  scanf("%lld", &a[i]);
        Mergesort(0,n - 1);                        //归并排序
        if(cnt <= k) printf("0\n");
        else            printf("%I64d\n", cnt - k);
    }
    return 0;
}
```

逆序对问题,除了可以用归并排序求解以外,也可以用树状数组求解。

6.2.2　快速排序

快速排序的思路是:把序列分成左、右两部分,使得左边所有的数都比右边的数小;递

归这个过程，直到不能再分为止。

那么如何把序列分成左、右两部分？最简单的办法是设定两个临时空间 X、Y 和一个基准数 t；检查序列中所有的元素，比 t 小的放在 X 中，比 t 大的放在 Y 中。其实不用这么麻烦，直接在原序列上操作就行了，不需要使用临时空间 X、Y。

直接在原序列上进行划分的方法也有很多种，下面的例子介绍了一种很容易操作的方法：

ij				*t*	
5	2	8	3	4	尾部的 t 是基准数，i 指向比 t 小的左部分，j 指向比 t 大的右部分。

i	*j*			*t*	
5	2	8	3	4	若 data[j]≥data[t]，j++。

	i	*j*		*t*	
2	5	8	3	4	若 data[j]<data[t]，交换 data[j]和 data[i]，然后 i++，j++。

		i	*j*	*t*	
2	3	8	5	4	继续。

		i	*j*	*t*	
2	3	4	5	8	最后，交换 data[i]和 data[t]，得到结果。i 指向基准数的当前位置。

下面用上述方法实现快速排序。

快速排序程序（poj 2388）

```
#include "stdio.h"
const int N = 10010;
int data[N];
#define swap(a, b) {int temp = a; a = b; b = temp;}      //交换
int partition(int left, int right){                   //划分成左、右两部分，以 i 指向的数为界
    int i = left;
    int temp = data[right];                           //把尾部的数看成基准数
    for(int j = left; j < right; j++)
        if(data[j] < temp){
            swap(data[j], data[i]);
            i++;
        }
    swap(data[i], data[right]);
     return i;                                        //返回基准数的位置
}
void quicksort(int left, int right){
    if(left < right){
        int i = partition(left, right);               //划分
          quicksort(left, i - 1);                     //分治：i 左边的继续递归划分
          quicksort(i + 1, right);                    //分治：i 右边的继续递归划分
    }
}
int main(){
```

```
    int n;
    scanf("%d", &n);
    for(int i=1; i<=n; i++)  scanf("%d", &data[i]);
    quicksort(1, n);
    printf("%d\n", data[(n+1)/2]);
    return 0;
}
```

下面分析**复杂度**。

每一次划分，都把序列分成了左、右两部分，在这个过程中，需要比较所有的元素，有 $O(n)$ 次。如果每次划分是对称的，也就是说左、右两部分的长度差不多，那么一共需要划分 $O(\log_2 n)$ 次。其总复杂度是 $O(n\log_2 n)$。

如果划分不是对称的，左部分和右部分的数量差别很大，那么复杂度会高一些。在极端情况下，例如左部分只有一个数，剩下的全部都在右部分，那么最多可能划分 n 次，总复杂度是 $O(n^2)$。所以，快速排序是不稳定的。


视频讲解

不过，一般情况下快速排序效率很高，甚至比稳定的归并排序更好。读者可以观察到，快速排序的代码比归并排序的代码简洁，代码中的比较、交换、复制操作很少。快速排序几乎是目前所有排序法中速度最快的方法。STL 的 sort()函数就是基于快速排序算法的，并针对快速排序的缺点做了很多优化。

快速排序思想可以用来解决一些特殊问题，例如求**第 k 大数**问题。

求第 k 大的数，简单的方法是用排序算法进行排序，然后定位第 k 大的数，其复杂度是 $O(n\log_2 n)$。

如果用快速排序的思想，可以在 $O(n)$ 的时间内找到第 k 大的数[①]。在快速排序程序中，每次划分的时候只要递归包含第 k 个数的那部分就行了。

【习题】

hdu 1425，求前 k 大的数；

poj 2388，求中间数。

6.3　减　治　法

大多数算法书籍不会特别讲解减治法(Decrease and Conquer)，减治法的题目常常被归纳到其他算法思想中。

用减治法解题的过程是把原问题分解为小问题，再把小问题分解为更小的问题，直到得到解。规模为 n 的原问题与分解后较小规模的子问题，它们的解有以下关系：

(1) 原问题的解只存在于其中一个子问题中；

(2) 原问题的解和其中一个子问题的解之间存在某种对应关系。

① 《算法导论》，Thomas H. Cormen，等著，潘金贵，等译，109 页，9.2 节。

按每次迭代中减去规模的大小把减治法分成以下 3 种情况：

(1) 减少一个常数。在算法的每次迭代中，把原问题减少相同的常数个，这个常数一般等于 1。相关的算法有插入排序、图搜索算法(DFS、BFS)、拓扑排序、生成排列、生成子集等。在这些问题中，每次把问题的规模减少 1。

(2) 按比例减少。在算法的每次迭代中，问题的规模按常数成倍减少，减少的效率极高。在大多数应用中，此常数因子等于 2。折半查找(Binary Search)是最典型的例子，在一个有序的数列中查找某个数 k，可以把数列分成相同长度的两半，然后在包含 k 的那部分继续折半，直到最后匹配到 k，总共只需要 $\log_2 n$ 次折半。

(3) 每次减少的规模都不同。减少的规模在算法的每次迭代中都不同，例如查找中位数(用快速排序的思路)、插值查找、欧几里得算法等。

6.4 小　　结

本章介绍了贪心、分治等基础算法的思想，这些也是算法竞赛中常见的题型。这两种算法思想容易理解、容易编程，若遇到难解的问题，大家不妨先考虑这两种方法。

第7章 动态规划

✍动态规划的概念和思想
✍最优子结构和重叠子问题
✍基础DP和递推法
✍0/1背包、LCS、LIS
✍滚动数组
✍记忆化搜索
✍区间DP
✍树形DP
✍数位DP
✍状态压缩DP

动态规划(Dynamic Programming,DP)题是算法竞赛中的必出题型。DP算法的效率高、代码少,竞赛队员不仅需要掌握很多编程技术,而且需要根据题目灵活设计具体的解题方案,能考察其思维能力、建模抽象能力、灵活性等。对DP的掌握情况很能体现竞赛队员的思维水平。

本章详细展开了与DP有关的算法,这些算法是每个竞赛队员都应该掌握的基本技术。

和贪心法、分治法一样,DP并不是一个特定的算法,而是一种算法思想。

DP算法思想可以简单解释如下:DP问题一般是多阶段决策问题,它把一个复杂问题分解为相对简单的子问题,再一个个解决,最后得到原复杂问题的最优解;这些子问题是前后相关的,并且非常相似,处理方法几乎一样。把前面子问题的计算结果记录为“状态”,并存储在“状态表”中,后面子问题可以直接查找前面得到的状态表,避免了重复计算,极大地减少了计算复杂度。

DP和分治法的区别如下:

(1) 分治法是把问题分成独立的子问题,各个子问题能独立解决,一个子问题内部的计算不需要其他子问题的数据,例如归并排序的分治过程。

(2) DP的子问题之间是相关的,前面子问题的解决结果被后面的子问题使用。

DP比分治法复杂得多。

DP适用于有重叠子问题和最优子结构性质的问题,具体的解释请参考算法类相关教材。

求解DP问题有3步,即定义状态、状态转移、算法实现。DP的核心是状态、状态转移方程。用状态转移方程求解状态,**状态往往就是问题的解**。在DP问题中,只要分析出状态以及状态转移方程,差不多就完成了90%的工作量。

DP问题可以分为线性和非线性的。

(1) 线性 DP。线性 DP 有两种方法,即顺推与逆推。在线性 DP 中,常常用“表格”来处理状态,用表格这种图形化工具可以清晰易懂地演示推导过程。本章绘制了大量表格来介绍有关算法。

(2) 非线性 DP。例如树形 DP,建立在树上,也有两个方向:①根→叶,根传递有用的信息给子结点,最后根得出最优解;②叶→根,根的子结点传递有用的信息给根,最后根得到最优解。

DP 是一种常用的算法思想。DP 问题可难可易,非常灵活,重点在于对“状态”和“转移”的建模与分析。该算法时间效率高,代码量少。在几乎所有的现场赛中都有 DP 的影子,而且常常作为中等题、难题出现。DP 一直是算法竞赛中的重点和难点。

7.1 基 础 DP

基础 DP 是一些经典问题,非常直观,易于理解。这些问题包括递推、0/1 背包、最长公共子序列、最长递增子序列等,它们的状态容易表示,转移方程容易得到。

下面从简单的硬币问题开始引导出动态规划的概念和处理方法。

7.1.1 硬币问题

前面第 6 章用贪心法解决的最少硬币问题要求硬币面值是特殊的。对于任意面值的硬币问题,需要用动态规划来解决。

硬币问题是简单的递推问题。

1. 最少硬币问题

有 n 种硬币,面值分别为 $v_1, v_2, \cdots, v_n$,数量无限。输入非负整数 s,选用硬币,使其和为 s。要求输出最少的硬币组合。

定义一个数组 int Min[MONEY],其中 Min[i]是金额 i 对应的最少硬币数量。如果程序能计算出 Min[i],$0<i<$MONEY,那么对输入的某个金额 i,只要查 Min[i]就得到了答案。

如何计算 Min[i]? Min[i]和 Min[$i-1$]是否有关系?

下面以 5 种面值(1、5、10、25、50)的硬币为例讲解递推的过程。

(1) 只使用最小面值的 1 分硬币。初始值 Min[0]=0,其他的 Min[i]为无穷大,如图 7.1 所示。下面计算 Min[1]。

金额i:	0	1	2	3	4	5	6	7	8	9 ...
硬币数量Min[]:	0	1								

图 7.1 只用 1 分硬币

$i=0$,Min[0]=0,表示金额为 0,硬币数量为 0。在这个基础上加一个 1 分硬币,就前进到金额 $i=1$、硬币数量 Min[1]=Min[0]+1=Min[1−1]+1= 1 的情况。

同理，$i=2$ 时，相当于在 Min[1]的基础上加一个硬币，得到 Min[2]=Min[2−1]+1=2。继续这个过程，结果如图 7.2 所示。

金额i:	0	1	2	3	4	5	6	7	8	9 ...
硬币数量Min[]:	0	1	2	3	4	5	6	7	8	...

图 7.2　只用 1 分硬币时的结果

分析上述过程，得到递推关系 Min[i]=min(Min[i]，Min[$i-1$]+1)。

(2) 在使用 1 分硬币的基础上增加使用第二大面值的 5 分硬币，如图 7.3 所示。此时应该从 Min[5]开始，因为比 5 分硬币小的金额不可能用 5 分硬币实现。

金额i:	0	1	2	3	4	5	6	7	8	9 ...
硬币数量Min[]:	0	1	2	3	4	1	6	7	8	...

图 7.3　加上 5 分硬币

$i=5$ 时，相当于在 $i=0$ 的基础上加一个 5 分硬币，得到 Min[5]=Min[5−5]+1=1。上一步用 1 分硬币的方案有 Min[5]=5。取最小值，得 Min[5]=1。

同理，$i=6$ 时，有 Min[6]=Min[6−5]+1=2，对比原来的 Min[6]=6，取最小值。

继续这个过程，结果如图 7.4 所示。

金额i:	0	1	2	3	4	5	6	7	8	9 ...
硬币数量Min[]:	0	1	2	3	4	1	2	3	4	...

图 7.4　加上 5 分硬币时的结果

递推关系是 Min[i]=min(Min[i]，Min[$i-5$]+1)。

(3) 继续处理其他面值的硬币。

在动态规划中，把 Min[i]这样的记录子问题最优解的数据称为“状态”，从 Min[$i-1$]或 Min[$i-5$]到 Min[i]的递推称为“状态转移”。用前面子问题的结果推导后续子问题的解，逻辑清晰、计算高效，这就是动态规划的特点。

程序代码如下：

```
#include <bits/stdc++.h>
using namespace std;
const int MONEY = 251;                    //定义最大金额
const int VALUE = 5;                      //5 种硬币
int type[VALUE] = {1, 5, 10, 25, 50};     //5 种面值
int Min[MONEY];                           //每个金额对应最少的硬币数量
void solve(){
    for(int k = 0; k < MONEY; k++)        //初始值为无穷大
        Min[k] = INT_MAX;
    Min[0] = 0;
    for(int j = 0; j < VALUE; j++)
        for(int i = type[j]; i < MONEY; i++)
```

```
            Min[i] = min(Min[i], Min[i - type[j]] + 1);    //递推式
}
int main(){
    int s;
    solve();                              //计算出所有金额对应的最少硬币数量,打表
    while(cin >> s)
        cout << Min[s] << endl;
    return 0;
}
```

solve()的复杂度是 O(VALUE×MONEY)。

需要注意的是,上面的 main()程序用到了"打表"的处理方法,即在输入金额之前提前用 solve()算出所有的解,得到 Min[MONEY]这个"表",然后再读取金额 s,查表直接输出结果,查一次表的复杂度只有 $O(1)$。这样做的原因是,如果有很多组测试数据,例如 10 000 个,那么总复杂度是 O(VALUE×MONEY+10 000),没有增加多少。如果不打表,每次读一个 s,就用 solve()算一次,那么总复杂度是 O(VALUE×MONEY×10 000),时间几乎多了 1 万倍。

2. 打印最少硬币的组合

在 DP 中,除求最优解的数量之外,往往还要求输出最优解本身,此时状态表需要适当扩展,以包含更多信息。

在最少硬币问题中,如果要求打印组合方案,需要增加一个记录表 Min_path[i],记录金额 i 需要的最后一个硬币。利用 Min_path[]逐步倒推,就能得到所有的硬币。

例如,金额 $i=6$,Min_path[6]=5,表示最后一个硬币是 5 分;然后,Min_path[6−5]=Min_path[1],查 Min_path[1]=1,表示接下来的最后一个硬币是 1 分;继续 Min_path[1−1]=0,不需要硬币了,结束。输出结果如图 7.5 所示,硬币组合是"5 分+1 分"。

金额i:	0	1	2	3	4	5	6	7	8	9...
Min[]:	0	1	2	3	4	1	2	3	4	...
Min_Path[]:	0	1	1	1	1	5	5	5	5	...

图 7.5　$i=6$ 时的输出结果

```
#include <bits/stdc++.h>
using namespace std;
const int MONEY = 251;                  //定义最大金额
const int VALUE = 5;                    //5 种硬币
int type[VALUE] = {1,5,10,25,50};       //5 种面值
int Min[MONEY];                         //每个金额对应最少的硬币数量
int Min_path[MONEY] = {0};              //记录最小硬币的路径

void solve(){
    for(int k = 0; k < MONEY;k++)
        Min[k] = INT_MAX;
```

```
    Min[0] = 0;
    for(int j = 0;j < VALUE;j++)
        for(int i = type[j]; i < MONEY; i++)
            if(Min[i] >  Min[i - type[j]] + 1){
                Min_path[i] = type[j];          //在每个金额上记录路径,即某个硬币的面值
                Min[i] = Min[i - type[j]] + 1; //递推式
            }
}
void print_ans(int * Min_path, int s) {         //打印硬币组合
    while(s){
        cout << Min_path[s] << " ";
        s = s - Min_path[s];
    }
}
int main() {
    int s;
    solve();
    while(cin >> s){
        cout << Min[s] << endl;                 //输出最少硬币个数
        print_ans(Min_path,s);                  //打印硬币组合
    }
    return 0;
}
```

3. 所有硬币组合

有 n 种硬币，面值分别为 $v_1, v_2, \cdots, v_n$，数量无限。输入非负整数 s，选用硬币，使其和为 s。输出所有可能的硬币组合。

> **hdu 2069 "Coin Change"**
>
> 有 5 种面值的硬币，即 1 分、5 分、10 分、25 分、50 分。输入一个钱数 s，输出组合方案的数量。例如 11 分有 4 种组合方案，即 11 个 1 分、2 个 5 分+1 个 1 分、1 个 5 分+6 个 1 分、1 个 10 分+1 个 1 分。$s \leqslant 250$，硬币数量 num$\leqslant$100。

如果用暴力法，可以逐个枚举各种面值的硬币个数，判断每种情况是否合法。枚举量是 $\frac{s}{50} \times \frac{s}{25} \times \frac{s}{10} \times \frac{s}{5} \times \frac{s}{1}$次。

1）不完全解决方案

假设硬币数量不限，即题目没有 num$\leqslant$100 的限制。

定义一个记录状态的数组 int dp[251]。dp[i]表示金额 i 所对应的组合方案数，即解空间。找到 dp[i]和 dp[$i-1$]的递推关系，就高效地解决了问题。

第一步：只用 1 分硬币进行组合。

dp[0]=1 为初始值。

dp[1]可以从 dp[0]推导出来：当金额 $s=1$ 时，如果用一个 1 分硬币，等价于从 s 中减去 1 分钱，并且硬币数量也减少一个的情况。此时退到 $i=0$；如果 $i=0$ 存在组合方案，那么 $i=1$ 的组合方案也存在。dp[1]=dp[1]+dp[0]。

对于其他 dp[i]，同样有 dp[i]=dp[i]+dp[$i-1$]。

在上述叙述中，dp[i]是"状态"，dp[i]=dp[i]+dp[$i-1$]是状态转移方程。前面子问题的状态 dp[$i-1$]，用状态转移方程计算后，得到后面子问题的状态 dp[i]。

计算可得表 7.1。

表 7.1 只用 1 分硬币时

i	0	1	2	3	4	5	6	7	8	…
dp[i]	1	1	1	1	1	1	1	1	1	…

第二步：加上 5 分硬币，继续进行组合。

当 $i<5$ 时，组合中不可能有 5 分硬币。

当 $i\geqslant 5$ 时，金额为 s 时的组合数量等价于从 s 中减去 5，而且硬币数量也减去一个的情况。dp[i]=dp[i]+dp[$i-5$]。计算可得表 7.2。

表 7.2 加上 5 分硬币时

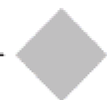
i	0	1	2	3	4	5	6	7	8	…
dp[i]	1	1	1	1	1	2	2	2	2	…

第三步：继续处理 10 分、25 分、50 分硬币的情况，同理有 dp[i]=dp[i]+dp[$i-10$]、dp[i]=dp[i]+dp[$i-25$]、dp[i]=dp[i]+dp[$i-50$]。

在上述步骤中，一次计算的复杂度只有 $O(1)$，全部计算的复杂度只有 $O(ks)$，k 是不同面值硬币的个数，s 是最大金额。

程序如下：

```
#include<bits/stdc++.h>
using namespace std;
const int MONEY = 251;                     //定义最大金额
int type[5] = {1, 5, 10, 25, 50};          //5 种面值
int dp[MONEY] = {0};
void solve() {
    dp[0] = 1;
    for(int i = 0;i<5;i++)
        for(int j = type[i]; j<MONEY; j++)
            dp[j] = dp[j] + dp[j - type[i]];
}
int main() {
    int s;
    solve();                               //提前计算出所有金额对应的组合数量,打表
    while(cin >> s)
        cout << dp[s] << endl;
    return 0;
}
```



2）完全解决方案

上述程序有一个问题，没有考虑对硬币数量的限制，hdu 2069 题要求硬币不能多于 100 个。这是因为状态 dp[*i*]太简单，没有记录计算过程中的细节。

重新定义状态为 dp[*i*][*j*]，建立一个“转移矩阵”，如表 7.3 所示。其中，横向是金额（题目中 $i\leqslant 250$），纵向是硬币数（题目中最多用 100 个硬币，$j\leqslant 100$）。

表 7.3　转移矩阵

j \ *i*	0	1	2	3	4	5	6	7	8	9	10	…
0	1											
1		1				1						
2			1				1				1	
3				1				1				…
4					1				1			
5						1				1		
6							1				1	
7								1				…
…									…			
100												

矩阵元素 dp[*i*][*j*]的含义是用 *j* 个硬币实现金额 *i* 的方案数量。例如表 7.3 中 dp[6][2]＝1，表示用两个硬币凑出 6 分钱，只有一种方案，即 5 分＋1 分。该表中的空格为 0，即没有方案，例如 dp[6][1]＝0，用一个硬币凑 6 分钱，不存在这样的方案。该表中列出了 dp[10][7]以内的方案数。

矩阵元素 dp[][]就是解空间。该表中纵坐标相加，就是某金额对应的方案总数，例如 6 分的金额为 dp[6][2]＋dp[6][6]＝2，有两种硬币组合方案。

“状态转移”的特征是用矩阵前面的状态 dp[*i*][*j*]能推算出后面状态的值。步骤如下：

第一步：只用 1 分硬币实现。

初始化：dp[0][0]＝1，其他为 0。定义 int type[5]＝{1，5，10，25，50}为 5 种硬币的面值。

从 dp[0][0]开始，可以推导后面的状态。例如，dp[1][1]是 dp[0][0]进行“金额＋1、硬币数量＋1”后的状态转移。转移后组合方案数量不变，即 dp[1][1]＝dp[0][0]＝1。

这里还要考虑 dp[1][1]原有的方案数，递推关系修正为：

dp[1][1]＝dp[1][1]＋dp[0][0]＝dp[1][1]＋dp[1－1][1－1]＝0＋1＝1

dp[1－1][1－1]的意思是从 1 分金额中减去 1 分硬币的钱，原来 1 个硬币的数量也减少 1 个。

在程序中，把上述操作写成：

dp[1][1]＝dp[1][1]＋dp[1－type[0]][1－1]

对所有 dp[i][j]进行上述操作，结果如表 7.4 所示。

表 7.4　只用 1 分硬币时

	0	1	2	3	4	5	6	7	8	9	10	…
0	1											
1		1										
2			1									
3				1								
4					1							
5						1						
6							1					
7								1				
…									…			
100												

第二步：加上 5 分硬币，继续进行组合。

dp[i][j]，当 $i<5$ 时，组合中不可能有 5 分硬币。

当 $i\geqslant 5$ 时，金额为 i、硬币为 j 个的组合数量等价于从 i 中减去 5 分钱，而且硬币数量也减去 1 个(即这个面值 5 的硬币)的情况。dp[i][j]＝dp[i][j]＋dp[$i-5$][$j-1$]＝dp[i][j]＋dp[$i-$type[1]][$j-1$]。对所有 dp[i][j]进行上述操作，结果如表 7.5 所示。

表 7.5　加上 5 分硬币时

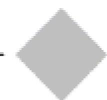
	0	1	2	3	4	5	6	7	8	9	10	…
0	1											
1		1				1						
2			1				1				1	
3				1				1				…
4					1				1			
5						1				1		
6							1				1	
7								1				…
…									…			
100												

第三步：陆续加上 10 分、25 分、50 分硬币，同理有以下关系。

$$\mathrm{dp}[i][j]=\mathrm{dp}[i][j]+\mathrm{dp}[i-\mathrm{type}[k]][j-1],k=2, 3, 4$$

总结上述过程，每个状态 dp[i][j]都可以根据它前面已经算出的状态进行推导，复杂度为 $O(1)$，总复杂度为 $O(kmn)$，k 是不同面值硬币的个数，m 和 n 是矩阵的大小。

利用 dp[i][j]也很容易找到某金额对应的最少和最多硬币数量。例如金额 5，最少硬币数量是 dp[5][1]＝1，最多硬币数量是 dp[5][5]＝5。



下面给出代码。

hdu 2069 代码

```
#include<bits/stdc++.h>
using namespace std;
const int COIN = 101;                       //题目要求不超过 100 个硬币
const int MONEY = 251;                      //题目给定的钱数不超过 250
int dp[MONEY][COIN] = {0};                  //DP 转移矩阵
int type[5] = {1, 5, 10, 25, 50};           //5 种面值
void solve() {                              //DP
    dp[0][0] = 1;
    for(int i=0; i<5; i++)
        for(int j=1; j<COIN; j++)
            for(int k = type[i]; k<MONEY; k++)
                dp[k][j] += dp[k-type[i]][j-1];
}
int main() {
    int s;
    int ans[MONEY] = {0};
    solve();                                //用 DP 计算完整的转移矩阵
    for(int i=0; i<MONEY; i++)              //对每个金额计算有多少种组合方案,打表
        for(int j=0; j<COIN; j++)           //从 0 开始,注意 dp[0][0]=1
            ans[i] += dp[i][j];
    while(cin>>s)
        cout<<ans[s]<<endl;
    return 0;
}
```

7.1.2　0/1 背包

0/1 背包是最经典的 DP 问题,没有之一。

背包问题:有多个物品,重量不同、价值不同,以及一个容量有限的背包,选择一些物品装到背包中,问怎么装才能使装进背包的物品总价值最大。根据不同的限定条件,可以把背包问题分为很多种,常见的有下面两种:

(1) 如果每个物体可以切分,称为一般背包问题,用贪心法求最优解。比如吃自助餐,在饭量一定的情况下,怎么吃才能使吃到肚子里的最值钱?显然应该从最贵的食物吃起,吃完了最贵的再吃第 2 贵的,这就是贪心法。

(2) 如果每个物体不可分割,称为 0/1 背包问题。仍以吃自助餐为例,这次食物都是一份份的,每一份必须吃完。如果最贵的食物一份就超过了你的饭量,那只好放弃。这种问题无法用贪心法求最优解。

1. 0/1 背包问题

给定 n 种物品和一个背包,物品 i 的重量是 w_i、价值为 v_i,背包的总容量为 C。在装入背包的物品时对每种物品 i 只有两种选择,即装入背包和不装入背包(称为 0/1 背包)。如何选择装入背包的物品,使得装入背包中的物品的总价值最大?

设 x_i 表示物品 i 装入背包的情况，当 $x_i=0$ 时不装入背包，当 $x_i=1$ 时装入背包，有以下约束条件和目标函数。

约束条件：

$$\sum_{i=1}^{n} w_i x_i \leqslant C \quad x_i \in \{0,1\}, \quad 1 \leqslant i \leqslant n$$

目标函数：

$$\max \sum_{i=1}^{n} v_i x_i$$



视频讲解

2. 用 DP 求解 0/1 背包

后面的描述都基于这个例子：有 4 个物品，其重量分别是 2、3、6、5，价值分别为 6、3、5、4，背包的容量为 9。

引进一个 $(n+1)\times(C+1)$ 的二维表 dp[][]，可以把每个 dp[i][j]都看成一个背包，dp[i][j]表示把前 i 个物品装入容量为 j 的背包中获得的最大价值，i 是纵坐标，j 是横坐标。

填表按照只装第 1 个物品、只装前两个物品、只装前 3 个物品的顺序，直到装完，如图 7.6 所示。这是从小问题扩展到大问题的过程。

背包容量	→	0	1	2	3	4	5	6	7	8	9
不装→	0										
装第 1 个→	1										
装前 2 个→	2										
装前 3 个→	3										
装前 4 个→	4										

图 7.6　填表的过程

步骤 1：只装第 1 个物品。

由于物品 1 的重量是 2，所以背包容量小于 2 的都放不进去，得 dp[1][0]=dp[1][1]=0；物品 1 的重量等于背包容量，装进去，背包价值等于物品 1 的价值，dp[1][2]=6；容量大于 2 的背包，多余的容量用不到，所以价值和容量 2 的背包一样，如图 7.7 所示。

		0	1	2	3	4	5	6	7	8	9
	0	0	0	0	0	0	0	0	0	0	0
$w_1=2, v_1=6$	1	0	0	*<u>6</u>*	6	6	6	6	6	6	6

图 7.7　只装第 1 个物品

步骤 2：只装前两个物品。

如果物品 2 的重量比背包容量大，那么不能装物品 2，情况和只装第 1 个物品一样。

下面填 dp[2][3]。物品 2 的重量等于背包容量，那么可以装物品 2，也可以不装：

(1) 如果装物品 2(重量是 3)，那么相当于只把物品 1 装到(容量−3)的背包中，如图 7.8 所示。

(2) 如果不装物品 2，那么相当于只把物品 1 装到背包中，如图 7.9 所示。

取(1)和(2)的最大值，得 dp[2][3]=max{3,6}=6。



		0	1	2	3	4	5	6	7	8	9
	0	0	0	0	0	0	0	0	0	0	0
$w_1=2, v_1=6$	1	0	0	6	6	6	6	6	6	6	6
$w_2=3, v_1=3$	2	0	0	6	***3+0***						

图 7.8　装物品 2

		0	1	2	3	4	5	6	7	8	9
	0	0	0	0	0	0	0	0	0	0	0
$w_1=2, v_1=6$	1	0	0	6	6	6	6	6	6	6	6
$w_2=3, v_1=3$	2	***0***	***0***	6	***6***						

图 7.9　不装物品 2

后续步骤：继续以上过程，最后得到图 7.10(图中的箭头是几个例子)。

		0	1	2	3	4	5	6	7	8	9
	0	0	0	0	0	0	0	0	0	0	0
$w_1=2, v_1=6$	1	0	0	6	6	6	6	6	6	6	6
$w_2=3, v_1=3$	2	0	0	6	6	6	9	9	9	9	9
$w_3=6, v_1=5$	3	0	0	6	6	6	9	9	9	***11***	11
$w_4=5, v_1=4$	4	0	0	6	6	6	9	9	***10***	11	11

图 7.10　最终结果

最后的答案是 dp[4][9]，即把 4 个物品装到容量为 9 的背包，最大价值是 11。

其算法复杂度是 $O(nC)$。

3. 输出 0/1 背包方案

现在回头看具体装了哪些物品，需要倒过来观察：

dp[4][9]=max{dp[3][4]+4,dp[3][9]}=dp[3][9]，说明没有装物品 4，用 $x_4=0$ 表示；

dp[3][9]=max{dp[2][3]+5,dp[2][9]}=dp[2][3]+5=11，说明装了物品 3，$x_3=1$；

dp[2][3]=max{dp[1][0]+3,dp[1][3]}=dp[1][3]，说明没有装物品 2，$x_2=0$；

dp[1][3]=max{dp[0][1]+6,dp[0][3]}=dp[0][1]+6=6，说明装了物品 1，$x_1=1$。

图 7.11 中的实线箭头指出了方案的路径。

		0	1	2	3	4	5	6	7	8	9	
	0	0	0	0	0	0	0	0	0	0	0	
$w_1=2, v_1=6$	1	0	0	6	***6***	6	6	6	6	6	6	$x_1=1$
$w_2=3, v_1=3$	2	0	0	6	***6***	6	9	9	9	9	9	$x_2=0$
$w_3=6, v_1=5$	3	0	0	6	6	6	9	9	9	11	***11***	$x_3=1$
$w_4=5, v_1=4$	4	0	0	6	6	6	9	9	10	11	***11***	$x_4=0$

图 7.11　查看具体装了哪些物品

4．例题

hdu 2602 "Bone Collector"

"骨头收集者"带着体积为 V 的背包去捡骨头，已知每个骨头的体积和价值，求能装进背包的最大价值。$N\leqslant 1000, V\leqslant 1000$。

输入：第 1 行是测试数量，第 2 行是骨头数量和背包体积，第 3 行是每个骨头的价值，第 4 行是每个骨头的体积。

1

5 10

1 2 3 4 5

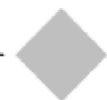
5 4 3 2 1

输出：最大价值。

14

代码如下：

```
#include<bits/stdc++.h>
using namespace std;
struct BONE{
    int val;
    int vol;
}bone[1011];
int T,N,V;
int dp[1011][1011];
int ans(){
    memset(dp,0,sizeof(dp));
    for(int i=1; i<=N; i++)
        for(int j=0; j<=V; j++){
            if(bone[i].vol > j)          //第 i 个物品太大,装不了
              dp[i][j] = dp[i-1][j];
            else                         //第 i 个物品可以装
            dp[i][j]=max(dp[i-1][j],
                      dp[i-1][j-bone[i].vol]+bone[i].val);
        }
    return dp[N][V];
}
int main(){
    cin>>T;
    while(T--){
        cin>>N>>V;
        for(int i=1;i<=N;i++)  cin>>bone[i].val;
        for(int i=1;i<=N;i++)  cin>>bone[i].vol;
        cout<<ans()<<endl;
    }
    return 0;
}
```

作为练习，请读者自己加上打印方案的程序。

5. 滚动数组

在处理 dp[][]状态数组的时候有一个小技巧：把它变成一维的 dp[]，以节省空间。观察上面的二维表 dp[][]可以发现，每一行是从上面一行算出来的，只跟上面一行有关系，跟更前面的行没有关系。那么用新的一行覆盖原来的一行就可以了。

hdu 2602（滚动数组程序）

```
int dp[1011];                              //替换 int dp[1011][1011];
int ans(){
    memset(dp, 0, sizeof(dp));
    for(int i = 1; i <= N; i++)
        for(int j = V; j >= bone[i].vol; j-- )  //反过来循环
            dp[j] = max(dp[j],dp[j - bone[i].vol] + bone[i].val);
    return dp[V];
}
```

注意，j 应该反过来循环，即从后面往前面覆盖。请读者思考原因。

经过滚动数组的优化，空间复杂度从 $O(NV)$ 减少为 $O(V)$。

动态规划题经常会给出很大的 N、V，此时需要使用滚动数组，否则会 MLE。

滚动数组也有缺点，它覆盖了中间转移状态，只留下了最后的状态，所以损失了很多信息，导致无法输出背包的方案。

【习题】

滚动数组请练习：

hdu 1024 “Max Sum Plus Plus”；

hdu 4576 “Robot”；

hdu 5119 “Happy Matt Friends”。

7.1.3 最长公共子序列

一个给定序列的子序列是在该序列中删去若干元素后得到的序列。例如 $X=(A,B,C,B,D,A,B)$，X 的子序列有(A,B,C,B,A)、(A,B,D)、(B,C,D,B)等。子序列和子串是不同的概念，子串的元素在原序列中是连续的。

给定两个序列 X 和 Y，当另一序列 Z 既是 X 的子序列又是 Y 的子序列时，称 Z 是序列 X 和 Y 的公共子序列。最长公共子序列是长度最长的子序列。

最长公共子序列(Longest Common Subsequence，LCS)问题：给定两个序列 X 和 Y，找出 X 和 Y 的一个最长公共子序列。

用暴力法找最长公共子序列需要先找出 X 的所有子序列，然后验证是否为 Y 的子序列。如果 X 有 m 个元素，那么 X 有 2^m 个子序列，若 Y 有 n 个元素，总复杂度大于 $O(n2^m)$。

用动态规划求 LCS，复杂度是 $O(nm)$。

例如，序列 $X=(a,b,c,f,b,c)$、$Y=(a,b,f,c,a,b)$，如图 7.12 所示。

用 $L[i][j]$ 表示子序列 X_i 和 Y_j 的最长公共子序列的长度。

当 $X_i=Y_j$ 时，找出 X_{i-1} 和 Y_{j-1} 的最长公共子序列，然后在其尾部加上 X_i 即可得到 X 和 Y 的最长公共子序列。

当 $X_i \neq Y_j$ 时，求解两个子问题：①求 X_{i-1} 和 Y_j 的最长公共子序列；②求 X_i 和 Y_{j-1} 的最长公共子序列。然后取其中的最大值。

$$L[i][j]=\begin{cases} L[i-1][j-1]+1 & X_i=Y_j, i>0, j>0 \\ \max\{L[i][j-1], L[i-1][j]\} & X_i \neq Y_j, i>0, j>0 \end{cases}$$

下面举例说明前几个步骤。

步骤 1：求 $L[1][1]$。有 $X_1=Y_1$，得 $L[1][1]=L[0][0]+1=1$，如图 7.13 所示。

	Y=	a	b	f	c	a	b
	0	1	2	3	4	5	6
X=	0						
a	1						
b	2						
c	3						
f	4						
b	5						
c	6						

图 7.12　序列 X 和 Y

	Y=	**a**	b	f	c	a	b
	0	1	2	3	4	5	6
X= 0	0	0	0	0	0	0	0
a 1	0	***1***					

图 7.13　求 $L[1][1]$

步骤 2：求 $L[1][2]$。有 $X_1 \neq Y_2$，得 $L[1][2]=\max\{L[1][1], L[0][2]\}=1$，如图 7.14 所示。

后续步骤：继续以上过程，最后得到图 7.15，$L[6][6]$ 就是答案。

	Y=	a	**b**	f	c	a	b
	0	1	2	3	4	5	6
X= 0	0	0	0	0	0	0	0
a 1	0	1 →	***1*** ↓				

图 7.14　求 $L[1][2]$

	Y=	a	b	f	c	a	b
	0	1	2	3	4	5	6
X= 0	0	0	0	0	0	0	0
a 1	0	1	1	1	1	1	1
b 2	0	1	2	2	2	2	2
c 3	0	1	2	2	3	3	3
f 4	0	1	2	3	3	3	3
b 5	0	1	2	3	3	3	4
c 6	0	1	2	3	4	4	4

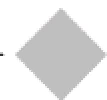
图 7.15　最终结果

如果要输出方案，和 0/1 背包的输出方案一样，LCS 需要从后面倒推回去。

下面给出一个例题。

hdu 1159 "Common Subsequence"

求两个序列的最长公共子序列。



代码如下：

```
#include <bits/stdc++.h>
using namespace std;
int dp[1005][1005];
string str1, str2;
int LCS(){
    memset(dp, 0, sizeof(dp));
    for(int i = 1; i <= str1.length(); i++)
        for(int j = 1; j <= str2.length(); j++){
            if(str1[i - 1]  ==  str2[j - 1])
                dp[i][j]  =  dp[i - 1][j - 1]  +  1;
            else
                dp[i][j]  =  max(dp[i - 1][j], dp[i][j - 1]);
        }
    return dp[str1.length()][str2.length()];
}
int main(){
    while(cin >> str1 >> str2)
        cout << LCS() << endl;
    return 0;
}
```

读者可以练习输出方案，并改为滚动数组。

7.1.4　最长递增子序列

最长递增子序列（Longest Increasing Subsequence，LIS）问题：给定一个长度为 N 的数组，找出一个最长的单调递增子序列。例如一个长度为 7 的序列 $A=\{5,6,7,4,2,8,3\}$，它最长的单调递增子序列为$\{5,6,7,8\}$，长度为 4。

下面给出一个例题。

hdu 1257 “最少拦截系统”

某国有一种导弹拦截系统，这种导弹拦截系统有一个缺陷：虽然它的第 1 发炮弹能够到达任意高度，但是以后每一发炮弹都不能超过前一发的高度。某天，雷达捕捉到敌国的导弹来袭，请计算最少需要多少套拦截系统。

输入：导弹总个数，导弹依此飞来的高度。

输出：最少要配备多少套这种导弹拦截系统。

输入样例：

8 389 207 155 300 299 170 158 65

输出样例：

2

这一题可以用贪心法做。假设发射了很多高度无穷大的导弹，在读入第 1 个炮弹时，一个导弹下降来拦截。以后每读入一个新的炮弹，都由能拦截它的最低的那个导弹来拦截。

最后统计用于拦截的导弹的个数,也就是最少需要的拦截系统的套数(请读者思考能否用这个贪心思想求最长递增子序列)。

下面用 DP 来做。

这个题目的思维转换有一些难度。题目的意思是统计一个序列中的单调递减子序列最少有多少个。这和最长递增子序列 LIS 有什么关系呢?

假设已经有了求 LIS 的算法,读者可能想这么做:把序列反过来,就变成了求反序列的递增子序列;先求反序列的第 1 个 LIS,然后从原序列中去掉这个 LIS,再对剩下的求第 2 个 LIS,直到序列为空;这些 LIS 的数量就是题目的解。

但是,其实并不用这么麻烦,这个题目实际上等价于求原序列的 LIS,这是一道求 LIS 的**裸题**,下面解释原因。

模拟计算过程:①从第 1 个数开始,找一个最长的递减子序列,即第 1 个拦截系统 X,在样例中是{389,300,299,170,158,65},去掉这些数,序列中还剩下{207,155};②在剩下的序列中再找一个最长递减子序列,即第 2 个拦截系统 Y,是{207,155}。

在 Y 中,至少有一个数 a 大于 X 中的某个数,否则 a 比 X 的所有数都小,应该在 X 中。所以,从每个拦截系统中拿出一个数能构成一个递增子序列,即拦截系统的数量等于这个递增子序列的长度。如果这个递增子序列不是最长的,那么可以从某个拦截系统中拿出两个数 c、d,在拦截系统中 $c>d$,c 和 d 不是递增的,这与递增序列的要求矛盾。

有多种方法可以求 LIS。

(1) 方法 1:上一节刚讲解了最长公共子序列 LCS,读者也许能想到借助 LCS。首先对序列排序,得到 A'={2,3,4,5,6,7,8},那么 A 的 LIS 就是 A 和 A'的 LCS。其复杂度是 $O(n^2)$。

(2) 方法 2:直接用 DP 求解 LIS。

定义状态 dp[i],表示以第 i 个数为结尾的最长递增子序列的长度,那么:

$\mathrm{dp}[i]=\max\{0,\mathrm{dp}[j]\}+1,\quad 0<j<i, A_j<A_i$

最后答案是 max{dp(i)}。

方法 2 的复杂度也是 $O(n^2)$,和方法 1 一样。

代码如下:

hdu 1257(DP 程序)

```
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 10000;
int n,high[MAXN];
int LIS(){
    int ans = 1;
    int dp[MAXN];
    dp[1] = 1;
    for(int i = 2; i <= n; i++){
        int max = 0;
        for(int j = 1; j < i; j++)
            if(dp[j] > max  &&  high[j] < high[i])
```



```
                max = dp[j];
            dp[i] = max + 1;
            if(dp[i] > ans) ans = dp[i];
        }
        return ans;
    }
    int main(){
        while(cin >> n){
            for(int i = 1; i <= n; i++)
                cin >> high[i];                    //输入炮弹高度值
            cout << LIS() << endl;
        }
        return 0;
    }
```

(3) 方法 3：有一种更快的、复杂度为 $O(n\log_2 n)$的方法。这个方法不是 DP 算法，它巧妙地利用了序列本身的特征，通过一个辅助数组 d[]统计最长递增子序列的长度。

定义：数组 d[]；len 统计 d[]内数据的个数；high[]为原始序列。

初始化：d[1]=high[1]；len=1；

操作步骤：逐个处理 high[]中的数字，例如处理到了 high[k]，①如果 high[k]比 d[]末尾的数字更大，就加到 d[]的后面；②如果 high[k]比 d[]末尾的数字小，就替换 d[]中第 1 个大于它的数字。

以 high[]={4,8,9,5,6,7}为例，表 7.6 所示为具体的操作过程。

表 7.6　方法 3 的具体操作过程

i	high[]	d[]	len	说　　明
1	**4**, 8, 9, 5, 6, 7	**4**	1	初始值 d[1]=high[1]
2	4, **8**, 9, 5, 6, 7	4 **8**	2	high[2]>d[1]，加到 d[]的后面
3	4, 8, **9**, 5, 6, 7	4 8 **9**	3	d[]后面加上 9
4	4, 8, 9, **5**, 6, 7	4 **5** 9	3	5 比 d[]末尾的 9 小，用 5 替换 d[]中第 1 个比 5 大的数 8
5	4, 8, 9, 5, **6**, 7	4 5 **6**	3	用 6 替换 9
6	4, 8, 9, 5, 6, **7**	4 5 6 **7**	4	d[]后面加上 7

结束后，len=4，就是 LIS 的长度。

为什么 d[]的长度等于 high[]的 LIS 的长度？分析算法对 high[]的两个关键操作：

(1)“如果 high[k]比 d[]末尾的数字更大，就加到 d[]后面”，high[]的 LIS 加 1，d[]的长度加 1，没有问题。

(2)“如果 high[k]比 d[]末尾的数字小，就替换 d[]中第 1 个大于它的数字”，有两个作用：首先，这个操作不影响 LIS 的长度，也不影响 d[]的长度；其次，high[]后面还没有处理的数很多都比已经处理过的数小，但是有可能序列更长，这里的替换给后面更小的数字留下了机会。为什么用 high[k]替换 d[]中第 1 个比它大的数字？因为数字 high[k]可能在 LIS 中，而被它替换的数字由于更大，不在 LIS 中的可能性更大。

在下面的代码中，对于“替换 d[]中第 1 个大于它的数字”这个功能，用 STL 的 lower_bound()函数帮助找到这个数，lower_bound()的复杂度是 $O(\log_2 n)$。程序的总复杂度是 $O(n\log_2 n)$[①]。

hdu 1257（非 DP 程序）

```
int LIS(){
    int len = 1;
    int d[MAXN];
    d[1] = high[1];                         //初始化
    for (int i = 2; i <= n; i++){           //O(n)
        if (high[i] > d[len])               //符合递增的要求,加入
            d[++len] = high[i];
        else{                               //替换
            int j = lower_bound(d + 1,d + len + 1,high[i]) - d;  //O(logn)
            d[j] = high[i];
        }
    }
    return len;
}
```

7.1.5 基础 DP 习题

(1) 简单题：

hdu 2018/2041/2044/2050/2182/4489。

(2) 背包：

有 0/1 背包、完全背包、分组背包、多重背包等。

hdu 1864 “最大报销额”，0/1 背包。

hdu 2159 “FATE”，完全背包。

hdu 2844 “Coins”，多重背包。

hdu 2955 “Robberies”，0/1 背包。

hdu 3092 “Least common multiple”，完全背包＋数论。

poj 1015 “Jury Compromise”。

poj 1170 “Shopping Offers”，状态压缩背包。

(3) LIS：

hdu 1003 “Max Sum”，最大连续子序列。

hdu 1087 “Super Jumping!”。

hdu 4352 “XHXJ's LIS”，数位 DP＋LIS。

poj 1239 “Increasing Sequence”，两次 dp。

(4) LCS：

hdu 1503 “Advanced Fruits”，LCS 变形。

① 最长不下降子序列是类似的问题。

poj 1080 "Human Gene Functions",LCS 变形。

7.2　递推与记忆化搜索

前面讲解 DP 的状态转移都是用递推的方法,另外还有一种方法,逻辑上的理解更加直接,这就是用"递归＋记忆化搜索"来实现 DP。

先看一道经典题。

> **poj 1163 "The Triangle"**
>
> 给定一个 n 层的三角形数塔,从顶部第 1 个数往下走,每层经过一个数字,直到最底层。注意,只能走斜下方的左边一个数或右边一个数。问所有可能走到的路径,最大的数字和是多少?

此题如果按"从顶往下"的计算方法,则由于可能有 2^n 个路径,导致 TLE。请读者思考为什么会有 2^n 个路径。

更快的方法是"从底往上"计算。按动态规划的思路,对数塔上的每个点记录状态,dp[i][j]记录从第 i 层第 j 个数开始往下走的数字和,每个结点算一次,一共有 $O(n^2)$个结点,所以复杂度是 $O(n^2)$。计算过程如图 7.16 所示,括号内的数字是 dp[i][j]:

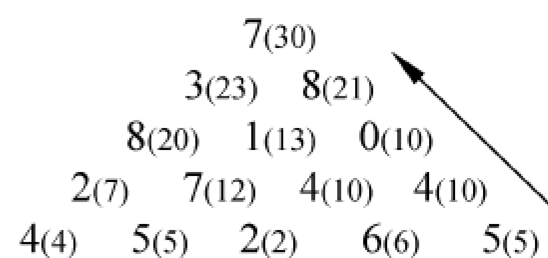


图 7.16　数塔

递推代码如下:

```
int a[150][150];                          //a[i][j]是数塔第 i 层的第 j 个数
int dp[150][150];                         //dp[i][j]记录从第 i 层第 j 个数开始往下走的数字和
for(int j = 1; j <= n;j++)  dp[n][j] = a[n][j];  //先计算最后一层
for(int i = n - 1;i >= 1;i -- )                  //从倒数第 2 层往上走到第 1 层
    for(int j = 1;j <= i;j++)                //从左边走上来,或者从右边走上来,取其中较大的
        dp[i][j] = a[i][j] + max(dp[i + 1][j], dp[i + 1][j + 1]);
```

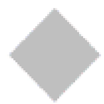
下面用"递归＋记忆化搜索"重新写 DP。

首先写出递归程序,搜索所有可能的路径。


视频讲解

```
int dfs(int i, int j) {
    if(i == n)
        return a[i][j];                  //递归边界:到达最后一行,返回
    return dp[i][j] = max(dfs(i + 1, j), dfs(i + 1, j + 1)) + a[i][j];
                                         //从左边走上来,或者从右边走上来,取其中较大的
}
```

这个 dfs()程序和前面"搜索技术"一章中讲解的 DFS 一样,是**暴力搜索**所有可能的情况。读者可以手工模拟递归过程,执行 dfs(1,1),程序一直递归到最底部的第 n 层,然后逐步回退,最后回到最顶部的第 1 层。最后的结果在 dp[1][1]中。dfs()的递归有 2^n 次,**暴力**



搜索了所有的 2^n 个路径，复杂度和前面最先提到的"从顶往下"的计算次数一样。

这个递归程序能优化吗？可以观察到，其中有大量重复计算，其实是能避免的。例如，观察图 7.16 中第 3 层的中间数"1"，从第 2 层的"3"往下走会经过"1"，计算一次从"1"出发的递归；从第 2 层的"8"往下走也会经过"1"，又重新计算了从"1"出发的递归。所以，只要避免这些重复计算就能优化。

下面的代码加上了"记忆化搜索"的内容：

```
//把 dp[][]初始化为 -1
int dfs(int i, int j) {
    if(i == n)
        return a[i][j];
    if(dp[i][j] >= 0)                          //记忆!如果计算过,就不再递归重算
        return dp[i][j];
    return dp[i][j] = max(dfs(i+1,j), dfs(i+1,j+1)) + a[i][j];
}
```

其中"if(dp[i][j] >= 0) return dp[i][j];"实现了"记忆化搜索"。

加上这一行代码后，如果发现 dp[i][j]已经计算过，就不再重算。由于数塔的结点有 $O(n^2)$个，每个点只需要计算一次 dp[i][j]，所以 dfs()的运行次数只有 $O(n^2)$次，和递推程序的复杂度一样。这样，就把暴力搜索的 $O(2^n)$次计算优化到了 $O(n^2)$次计算。记忆化搜索的优化能力是惊人的。

记忆化搜索。在用递归实现 DP 时，在递归程序中记录计算过的状态，并在后续的计算中跳过已经算过的重复的状态，从而大大减少递归的计算次数，这就是"记忆化搜索"的思路。

在很多情况下，"记忆化搜索"的逻辑思路和程序比直接写递推更简单。在本书"7.5 数位 DP"中有相关的例子。

7.3 区间 DP

区间 DP 的主要思想是先在小区间进行 DP 得到最优解，然后再利用小区间的最优解合并求大区间的最优解。

区间 DP，一般需要从小到大枚举所有可能的区间。在解题时，先解决小区间问题，然后合并小区间，得到更大的区间，直到解决最后的大区间问题。合并的操作一般是把左、右两个相邻的子区间合并。

区间 DP 的两个难点：枚举所有可能的区间、状态转移方程。

区间 DP 的复杂度：一个长度为 n 的区间，它的子区间数量级为 $O(n^2)$，每个子区间内部处理时间不确定，合起来复杂度会大于 $O(n^2)$。在编程时，区间 DP 至少需要两层 for 循环，第 1 层的 i 从区间的首部或尾部开始，第 2 层的 j 从 i 开始到结束，i 和 j 一起枚举出所

有的子区间。例如：

```
for(int i = 1;i < n;i++)                  //n是区间长度
    for(int j = i;j <= n;j++)             //j每次递增1,也可能跨步递增
        …
```

下面用两个经典问题讲解。

1. 石子合并

> **"石子合并"**
>
> 有 n 堆石子排成一排，每堆石子有一定的数量，将 n 堆石子合并成一堆。合并的规则是每次只能合并相邻的两堆石子，合并的花费为这两堆石子的总数。石子经过 $n-1$ 次合并后成为一堆，求总的最小花费。
>
> 输入：有多组测试数据，输入到文件结束。每组测试数据的第 1 行有一个整数 n，表示有 n 堆石子，$n<250$。接下来的一行有 n 个数，分别表示这 n 堆石子的数目。每堆石子至少 1 颗，最多 10 000 颗。
>
> 输出：总的最小花费。
>
> 输入样例：
>
> 3
>
> 2 4 5
>
> 输出样例：
>
> 17

样例的计算过程是：①第一次合并 $2+4=6$；②第二次合并 $6+5=11$；总花费是 $6+11=17$。

DP 的状态如何设计？设 dp[i][j]为从第 i 堆石子到第 j 堆石子的最小花费，那么 dp[1][n]就是答案。另外，设 sum[i][j]为从第 i 到 j 的区间的和。

为了计算最后的 dp[1][n]，需要考虑所有可能的合并。这些合并包括：

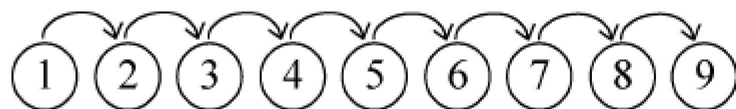


图 7.17 两堆合并

(1) 合并之前，dp[i][i]$=0$，$1\leqslant i\leqslant n$。

(2) 两堆合并，如图 7.17 所示。

例如：dp[1][2]=dp[1][1]+dp[2][2]+sum[1][2]；

总结：dp[i][$i+1$]=dp[i][i]+dp[$i+1$][$i+1$]+sum[i][$i+1$]；

(3) 三堆合并，如图 7.18 所示。

例如合并第 1 堆到第 3 堆，有两种情况，如图 7.19 所示。



图 7.18 三堆合并



图 7.19 两种情况

$dp[1][3]=\min(dp[1][1]+dp[2][3],dp[1][2]+dp[3][3])+sum[1][3]$;

总结：$dp[i][i+2]=\min(dp[i][i]+dp[i+1][i+2],dp[i][i+1]+dp[i+2][i+2])+sum[i][i+2]$;

(4) 推广：第 i 堆到第 j 堆的合并，如图 7.20 所示。

$dp[i][j]=\min(dp[i][k]+dp[k+1][j])+sum[i][j-i+1]$，$i\leqslant k\leqslant j$。这就是状态转移方程。

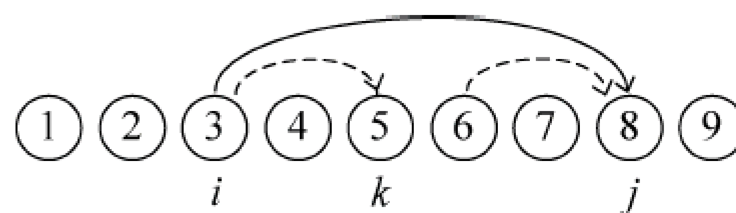


图 7.20 第 i 堆到第 j 堆的合并

下面的函数 Minval()实现了上述计算过程，其中有 3 层循环：

(1) 最外面一层的变量 len 表示区间$[i,j]$的长度，从 2 到 n；

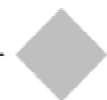
(2) 第二层枚举的起点位置 i 从 1 到 $n-$len，终点通过计算得到，$j=i+$len；

(3) 在区间$[i,j]$里枚举每个分割的位置 k。

虽然下面的代码很短，但是逻辑比较复杂，请读者仔细体会并能自己写出来。

```cpp
#include <bits/stdc++.h>
using namespace std;
const int INF = 1 << 30;
const int N = 300;
int sum[N], n;
int Minval() {
    int dp[N][N];
    for(int i = 1; i <= n; i++)
        dp[i][i] = 0;
    for(int len = 1; len < n; len++)              //len 是 i 和 j 之间的距离
        for(int i = 1; i <= n - len; i++) {       //从第 i 堆开始
            int j = i + len;                      //到第 j 堆结束
            dp[i][j] = INF;
            for(int k = i; k < j; k++)            //i 和 j 之间用 k 进行分割
                dp[i][j] = min(dp[i][j],
                      dp[i][k] + dp[k + 1][j] + sum[j] - sum[i - 1]);
        }
    return dp[1][n];
}
int main() {
    while(cin >> n) {
        sum[0] = 0;
        for(int i = 1; i <= n; i++) {
            int x;
            cin >> x;
            sum[i] = sum[i - 1] + x;              //sum[i, j]的值等于 sum[j] - sum[i - 1]
        }
        cout << Minval() << endl;
    }
    return 0;
}
```

复杂度：Minval()中有三重循环，复杂度是 $O(n^3)$。所以上述算法只能用来处理规模 $n<250$ 的问题。



那么 Minval()是否可以优化？在它的三重循环中，前两重循环是枚举所有可能的合并，无法优化，最后一层循环枚举分割点 k，是可以优化的。因为每次运行最后一层循环时都在某个子区间内部寻找最优分割点，该操作在多个子区间里是重复的。如果找到这个最优点后保存下来，用于下一次循环，就能避免重复计算，从而降低复杂度。

用 $s[i][j]$ 表示区间 $[i,j]$ 中的最优分割点，第三重循环可以从区间 $[i,j-1)$ 的枚举优化到区间 $[s[i][j-1], s[i+1][j]]$ 的枚举。其中，$s[][]$ 值是在前面的第三重循环中找到并记录下来的。

上述讨论符合“平行四边形优化”的原理，它是区间 DP 的常见优化方法。请读者自行了解并掌握。

经过优化以后，复杂度接近 $O(n^2)$，可以解决 $n<3000$ 的问题。上面的程序只需要修改 3 处，在下面的代码中使用斜体显示：

```
int Minval() {
    int dp[N][N], s[N][N];
    for(int i = 1; i <= n; i++){
        dp[i][i] = 0;
        s[i][i] = i;                              //初始值
    }
    for(int len = 1; len < n; len++)
        for(int i = 1; i <= n - len; i++)  {
            int j = i + len;
            dp[i][j] = INF;
            for(int k = s[i][j - 1]; k <= s[i + 1][j]; k ++ )  //缩小范围
                if(dp[i][k] + dp[k + 1][j] + sum[j] - sum[i - 1]< dp[i][j]){
                    dp[i][j] = dp[i][k] + dp[k + 1][j] + sum[j] - sum[i - 1];
                    s[i][j] = k;                  //记录[i, j]的最优分割点
                }
        }
    return dp[1][n];
}
```

2. 回文串

回文串是正读和反读都一样的字符串，例如"abcdcba"。回文串问题是经典的字符串问题：给定一个字符串，然后通过增加或删除部分字符串得到一个回文串。

poj 3280 “Cheapest Palindrome”

给定字符串 s，长度为 m，由 n 个小写字母构成。在 s 的任意位置增删字母，把它变为回文串，增删特定字母的花费不同。求最小花费。

输入样例：

3 4
abcb
a 1000 1100
b 350 700
c 200 800

输出样例：
900

输入的第 1 行是 n 个字符，长度为 m；第 2 行是字符串 s，后面 n 行分别给出每个字符插入和删除的花费。

在样例中，如果在结尾处插入"a"，得到"abcba"，花费 1000；如果在首端删除"a"得到"bcb"，花费 1100；如果在首端插入"bcb"，花费 350+200+350=900，这是最小值。

该题有多种解法，其中一种是把 s 反转得到 s'，求得两者最长公共子序列的长度 l，用 s 的长度减去 l 就是答案。

下面用区间 DP 的方法求解。

定义状态 dp[i][j]表示字符串 s 的子区间 s[i,j]变成回文的最小花费。

另外，在考虑删除和插入的花费时，由于这两种操作是等价的（这头加和那头减一样），所以只要取这两种操作的最小值就行了。用数组 w[]定义字符的花费。

有以下 3 种情况：

（1）如果 $s[i]==s[j]$，那么 dp[i][j]=dp[i+1][j−1]，如图 7.21 所示。

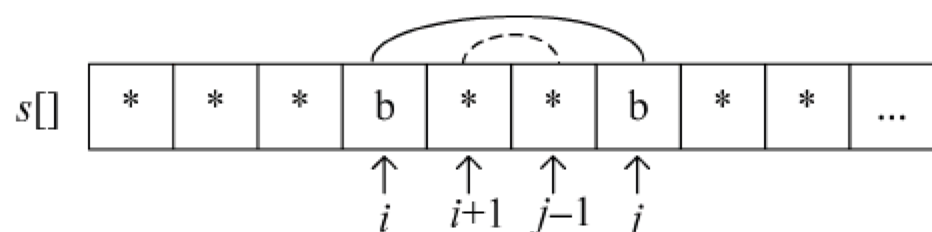


图 7.21　情况 1

（2）如果 dp[i+1][j]是回文串，那么 dp[i][j]=dp[i+1][j]+w[i]，如图 7.22 所示。

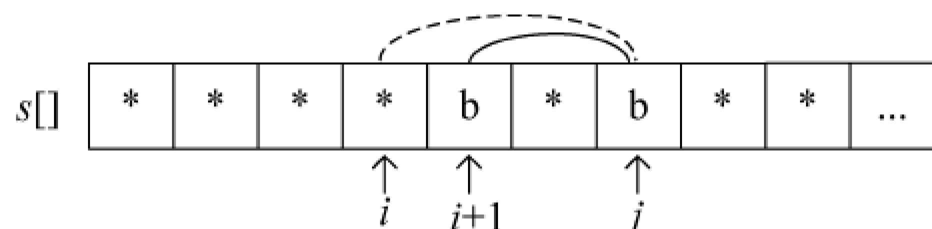


图 7.22　情况 2

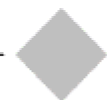
（3）如果 dp[i][j−1]是回文串，那么 dp[i][j]=dp[i][j−1]+w[j]。

总结情况 2、3，状态转移方程是 dp[i][j]=min(dp[i+1][j]+w[i]，dp[i][j−1]+w[j])。

该程序中包含两层循环，外层 i 枚举子区间起点，内层 j 枚举终点。因为需要从小区间扩展到大区间，所以 i 从 s 的尾端开始，逐步回退扩大区间，直到首端。

poj 3280 程序

```
#include <iostream>
using namespace std;
int w[30],n,m,dp[2005][2005];
char s[2005], ch;
int main() {
    int x,y;
    while(cin >> n >> m) {                          //n 是用到的字符个数,m 是 s 的长度
```



```
        cin >> s;
        for(int i = 0;i < n;i++) {
            cin >> ch >> x >> y;                    //读取每个字符的插入和删除的花费
            w[ch - 'a'] = min(x,y);                 //取其中的最小值
        }
        for(int i = m - 1; i >= 0; i -- )            //i 是子区间的起点
            for(int j = i + 1; j < m; j++) {         //j 是子区间的终点
                if(s[i] == s[j])
                    dp[i][j] = dp[i + 1][j - 1];
                else
                    dp[i][j] = min(dp[i + 1][j] + w[s[i] - 'a'],
                                   dp[i][j - 1] + w[s[j] - 'a']);
            }
            cout << dp[0][m - 1]<< endl;
    }
    return 0;
}
```

【习题】

hdu 3506 “Monkey Party”,环形石子合并。

hdu 4283 “You Are the One”,区间 DP。

hdu 4632 “Palindrome Subsequence”,回文串。

hdu 2476 “String Printer”,区间 DP。

hdu 4745 “Two Rabbits”,最长回文子序列。

hdu 5115 “Dire Wolf”,区间 DP。

poj 1141 “Brackets Sequence”,括号匹配。

poj 2955 “Brackets”,区间 DP。

7.4　树　形　DP

树形 DP 是指在“树”这种数据结构上进行的 DP：给出一棵树,要求以最少的代价(或取得最大收益)完成给定的操作。通常这类问题规模较大,枚举算法的效率低,无法胜任,贪心算法不能得到最优解,因此需要用动态规划。

在树上做动态规划显得非常合适,因为树本身有“子结构”性质(树和子树),具有递归性,符合 DP 的性质。相比线性 DP,树形 DP 的状态转移方程更加直观。不过,由于“树”这种数据结构比较烦琐,逻辑上比较复杂,状态转移方程不好设计,常常属于比较难的题目。

树的操作一般需要利用递归和搜索,用户需要熟练地掌握这些基础知识。树的遍历一般是从根结点往子结点方向深入,用 DFS 编程会比较简单。

下面从一个最基础的树形 DP 开始。

hdu 1520 "Anniversary Party"

一棵有根树上每个结点有一个权值，相邻的父结点和子结点只能选择一个，问如何选择使得总权值之和最大(邀请员工参加宴会，为了避免员工和直属上司发生尴尬，规定员工和直属上司不能同时出席)。

输入：结点编号从 1 到 N。第 1 行是一个数字 N，$1 \leqslant N \leqslant 6000$。后续 N 行中的每一行都包含结点的权值，范围是 -128 到 127 的整数。下面是 T 行，描述一个父子关系，每一行都有如下形式：

L K

第 K 个结点是第 L 个结点的父结点。读到 0 0 时结束。

输出：输出总的最大权值。

输入样例：

```
5
1
1
1
1
1
1 3
2 3
4 5
3 5
0 0
```

输出样例：

```
3
```

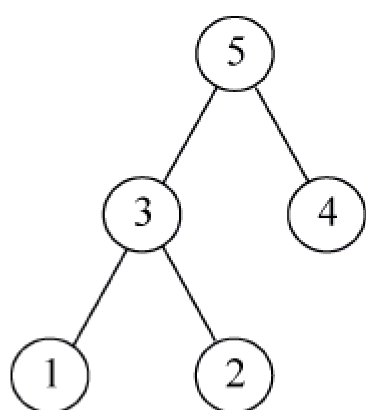


图 7.23　样例的树形关系

图 7.23 是样例的树结构。当结点选 1、2、5 时有最大值 3。读者可以思考，如果用暴力的方法遍历所有的情况，复杂度是多少。

根据 DP 的解题思路，定义状态为：

dp[i][0]，表示不选择当前结点时的最优解；

dp[i][1]，表示选择当前结点时的最优解。

状态转移方程有两种情况：

(1) 不选择当前结点，那么它的子结点可选可不选，取其中的最大值：

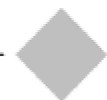
dp[u][0]+=max(dp[son][1], dp[son][0])

(2) 选择当前结点，那么它的子结点不能选，dp[u][1]+=dp[son][0]。

程序包含 3 个部分：

(1) 建树。本题可以用 STL 的 vector 生成链表，建立关系树。

(2) 树的遍历。可以用 DFS，从根结点开始进行记忆化搜索。



(3) DP。

```
#include <bits/stdc++.h>
using namespace std;
const int N = 6000+5;
int value[N], dp[N][2], father[N], n;
vector<int> tree[N];
void dfs(int u){
    dp[u][0] = 0;                                   //赋初值:不参加宴会
    dp[u][1] = value[u];                            //赋初值:参加宴会
    for(int i=0;i<tree[u].size();i++){             //逐一处理这个父结点的每个子结点
        int son = tree[u][i];
        dfs(son);                                   //深搜子结点
        dp[u][0] += max(dp[son][1], dp[son][0]);
                                                    //父结点不选,子结点可选可不选
        dp[u][1] += dp[son][0];                     //父结点选择,子结点不选
    }
}
int main(){
    while(~scanf("%d",&n)) {
      for(int i=1;i<=n;i++)  {
          scanf("%d",&value[i]);
          tree[i].clear();
          father[i] = -1;                           //赋初值,还未建立关系
      }
      while(1) {
        int a,b;
        scanf("%d%d",&a,&b);
        if(a==0&&b==0)  break;
        tree[b].push_back(a);                       //用邻接表建树
        father[a]=b;                                //父子关系
      }
      int t = 1;
      while(father[t] != -1)                        //查找树的根结点
        t = father[t];
      dfs(t);                                       //从根结点开始,用 DFS 遍历整棵树
      printf("%d\n", max(dp[t][1], dp[t][0]));
    }
    return 0;
}
```

复杂度:上述代码遍历每个结点,总复杂度是 $O(n)$。

下面是一个中等难度的树形 DP 的题目,逻辑和状态转移都比较复杂。

hdu 2196 "Computer"

一棵有根树,根结点的编号是 1,对其中的任意一个结点,求离它最远的结点的距离。

输入：输入文件包含多个测试用例。每个用例的第 1 行是一个自然数 $N(N\leqslant 10\,000)$，后面有 $N-1$ 行。第 i 行包含两个自然数：某个结点；第 i 个结点连接到这个结点的距离，距离长度不超过 10 ^9。

输出：输出 N 行。第 i 行是距离第 i 个结点的最远距离。

输入样例：

```
5
1 1
2 1
3 1
1 1
```

输出样例：

```
3
2
3
4
4
```

复杂度分析：如果求从一个特定结点出发的最长路径，可以从这个结点出发，做一次 BFS，每次扩展邻居结点，并记录到这个邻居结点的最长距离，复杂度是 $O(n)$。求所有 n 个结点的最长距离，需要对每个结点单独做一次 BFS，总复杂度是 $O(n^2)$。但是由于题目规模较大，$N\leqslant 10\,000$，所以算法的复杂度最多只能是 $O(n\log_2 n)$。下面用动态规划求解。

一棵有根树如图 7.24 所示。

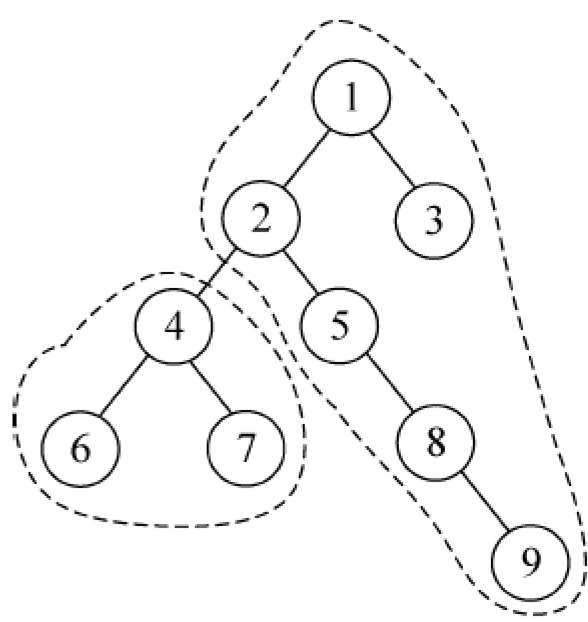


图 7.24　一棵有根树

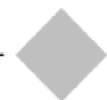
以结点 4 为例，它的最长距离分两种情况讨论：

(1) 以 4 为顶点的子树(图 7.24 左边圈起的部分)距结点 4 的最远距离 L_1。对结点 4 来说，它的 L_1 很容易求，只要从结点 4 出发对它的子树做一次 DFS，记录最大深度，就能求得 L_1。那么如何求得树上所有结点的 L_1 值？可以从根结点 1 开始 DFS，遍历所有的结点，在 DFS 返回的过程中记录每个结点的最大深度，即这个结点的 L_1 值(在下面的程序中实际上计算了每个结点的两个距离。以结点 4 为例，这两个距离是以 4 为顶点的最长距离 one，即 L_1 值；以 4 为顶点的第 2 长距离 two)。

(2) 剩下部分(图 7.24 右边圈起的部分)到结点 4 的最长距离 L_2。L_2＝父结点 2 的最长距离＋ dist(2,4)，dist(2,4)是 2 和 4 之间的距离。求 L_2 的关键是求父结点 2 的最长距离，它又分两种情况：

① 从结点 2 往上走的最长距离，图中路径是 2-1-3。这可以通过 DFS 不断更新结点来获得，具体操作见下面的程序。

② 结点 2 除了结点 4 以外的其他子树的最长距离 X，图中路径是 2-5-8-9。在上面第



(1)步中已经求得了每个结点的最长距离 one 和次长距离 two。如果结点 4 在父结点 2 的最长子树上，那么 $X=\text{two}+\text{dist}(2,4)$；如果结点 4 不在父结点 2 的最长子树上，那么 $X=\text{one}+\text{dist}(2,4)$。

综上所述，距离结点 4 最远的距离是 $\max\{L_1,L_2\}$。在程序中，用 dfs1()实现功能(1)，用 dfs2()实现功能(2)。

状态的设计：结点 i 的子树到 i 的最长距离 dp[i][0]以及次长距离 dp[i][1]；从结点 i 往上走的最短距离 dp[i][2]。

```
#include<bits/stdc++.h>
using namespace std;
const int N = 10100;
struct Node{
    int id;                                 //子结点的编号
    int cost;
};
vector<Node> tree[N];
int dp[N][3];
int n;
void init_read(){
    for(int i=1; i<=n; i++)
        tree[i].clear();
    memset(dp, 0, sizeof(dp));
    for(int i=2; i<=n; i++) {
        int x,y;
        scanf("%d%d",&x,&y);
        Node tmp;
        tmp.cost = y;
        tmp.id=i;                           //i 是 x 的子结点
        tree[x].push_back(tmp);
    }
}
void dfs1(int father) {                     //DFS,先处理子结点,再处理父结点
    int one=0, two=0;
    for(int i=0; i<tree[father].size(); i++) {
                                            //遍历结点 father 的所有子结点
        Node child = tree[father][i];
        dfs1(child.id);                     //递归子结点,直到最底层
        int cost = dp[child.id][0] + child.cost;
        if(cost >= one) {                   //用 one 记录从 father 往下走的最长距离
            two = one;                      //原来的最长距离 one 变成第 2 长,用 two 记录
            one = cost;
        }
        if(cost < one && cost > two)        //用 two 记录第 2 长的距离
            two = cost;
    }
    dp[father][0] = one;                    //得到以 father 为起点的子树的最长距离
    dp[father][1] = two;                    //得到以 father 为起点的子树的第 2 长距离
}
void dfs2(int father) {                     //先处理父结点,再处理子结点
```

```
    for(int i = 0; i < tree[father].size(); i++) {
        Node child = tree[father][i];
        if(dp[child.id][0] + child.cost == dp[father][0])
                                    //child 在最长距离的子树上
            dp[child.id][2] = max(dp[father][2], dp[father][1]) + child.cost;
        else                                //child 不在最长距离的子树上
            dp[child.id][2] = max(dp[father][2], dp[father][0]) + child.cost;
        dfs2(child.id);
    }
}
int main(){
    while(~scanf("%d", &n)) {
        init_read();                        //初始化,读数据
        dfs1(1);                            //计算 dp[][0]、dp[][1]
        dp[1][2] = 0;
        dfs2(1);                            //计算 dp[][2]
        for(int i = 1; i <= n; i++)
            printf("%d\n", max(dp[i][0], dp[i][2]));
    }
    return 0;
}
```

复杂度：dfs1()和 dfs2()的复杂度约为 $O(n)$。

【习题】

下面题目的难度都在中等以上。

poj 2378/3107/3140；

hdu 1011/1561/2242/3586/5834。

7.5 数 位 DP

先从一道简单题引出数位 DP 的概念。

> **hdu 2089 "不要 62"**
>
> 一个数字，如果包含'4'或者'62'，它是不吉利的。给定 m 和 n，$0<m\leqslant n<10^6$，统计 $[m, n]$ 范围内吉利数的个数。

这一题的数据范围是 10^6，但是此类题目常常达到 10^{18}。暴力方法是检查每一个数，复杂度大于 $O(n)$。由于 n 太大，肯定会超时，需要设计一个时间复杂度接近 $O(\log_2 n)$的算法。

读者很容易想到排除法。基本思路是在 $0\sim10^6$ 内排除不符合条件的数，具体操作是按"从高位到低位"的顺序进行判断。例如，求 1～999 999 内不包含 4 的数(对数字'62'的处理方法类似)，步骤如下：

(1) 在 6 位数中排除最高位是 4 的数,即 400 000～499 999。虽然有 10 万个数,但只需要判断最高位,一次就全排除了。

(2) 在最高位不是 4 的 6 位数中排除次高位是 4 的数。例如最高位是 1 的数可以一次性排除 140 000～149 999,共 1 万个。注意,首位**可以是 0**,即 000 000～099 999 也算 6 位数。

(3) 继续排除 5 位数、4 位数等,直到结束。

下面用数位 DP 的方法实现上述排除法的思路。

所谓“数位 DP”,是指对数字的“位”进行的与计数有关的 DP。一个数有个位、十位、百位、千位等,数的每一位就是数位。数位 DP 用来解决与数字操作有关的问题,例如数位之和问题、特定数字问题等。这些问题的特征是给定的区间超级大,不能用暴力的方法逐个检查,必须用接近 $O(\log_2 n)$复杂度的算法。解题的思路是用 DP 对“数位”进行操作,记录已经算过的区间的状态,用在后续计算中,快速进行大范围的筛选。

回头考虑 hdu 2089 题,统计所有的吉利数,用 DP 怎么做? 下面用两种方法实现 DP,一种用递推公式,一种用记忆化搜索。

1. 用递推实现 hdu 2089 题

定义状态 dp[i][j],它表示 i 位数中首位是 j,符合要求的数的个数。例如 dp[6][1]表示首位是 1 的 6 位数,即 100 000～199 999 中符合要求的数有多少个。那么如何求 dp[6][1]? 计算首位数字 1 后面的 5 位数就可以了,即计算 00 000～99 999 中符合要求的数。所以,dp[i][j]的递推公式如下:

$$\mathrm{dp}[i][j] = \sum_{k=0}^{9} \mathrm{dp}[i-1][k], \quad (j \neq 4)\&\&(k \neq 2\&\&j \neq 6)$$

下面是程序。为了突出对数位 DP 思路的理解,程序简化了题目的要求,只排除了'4',没有排除"62"。作为练习,读者可以自己加上对"62"的处理。从下面的程序可知,求 dp[i][j]的计算复杂度极小,i、j、k 的三重循环只需要计算 1000 次。

统计[0,n]内不含 4 的数字个数(递推程序)

```
#include <bits/stdc++.h>
const int LEN = 12;              //可以更大
int dp[LEN][10];                 //dp[i][j]表示 i 位数,第 1 个数是 j 时符合条件的数字数量
int digit[LEN];                  //digit[i]存第 i 位数字
void init(){
    dp[0][0] = 1;
    for(int i = 1; i <= LEN; i++)
        for(int j = 0; j < 10; j++)
            for(int k = 0; k < 10; k++)
                if(j!= 4)            //排除数字 4
                    dp[i][j] += dp[i - 1][k];
}
int solve(int len) {                 //计算[0,n]区间满足条件的数字个数
    int ans = 0;
    for(int i = len; i >= 1; i -- ){  //从高位到低位处理
        for(int j = 0; j < digit[i]; j++)
            if(j!= 4)
```

```
                ans += dp[i][j];
            if(digit[i] == 4) {           //第 i 位是 4,以 4 开头的数都不行
              ans -- ; break;}
        }
        return ans;
    }
    int main(){
        int n,len = 0;
        init();                           //预计算 dp[][]
        scanf("%d",&n);
        while(n){                         //len 是 n 的位数.例如 n = 324,是 3 位数,len = 3
            digit[++len] = n%10;
                                          //例如 n = 324,digit[3] = 3, digit[2] = 2, digit[1] = 4
            n/ = 10;
        }
        printf("%d\n",solve(len) + 1);    //求[0,n]内不含 4 的数字个数
        return 0;
    }
```

程序中的 init()是预处理,求 dp[][],如表 7.7 所示(这里只画了部分箭头)。

表 7.7　dp[i][j]的计算

j \ i	0	1	2	3	4	5	6	7	8	9	10	…
0	1	1	9	81	729	…						
1		1	9	81	729	…						
2		1	9	81	729	…						
3		1	9	81	729	…						…
4		0	0	0	0	…						
5		1	9	81	729	…						
6		1	9	81	729	…						
7		1	9	81	729	…						…
8		1	9	81	729	…						
9		1	9	81	729	…						

然后用 solve()完成题目的计算。题目要求计算给定范围内符合要求的数,那么把相应的 dp[i][j]相加即可(加的时候需要判断'4')。例如,求[0,324]内符合条件的数,设答案是 ans,计算步骤如下:

(1) 处理 3 位数,ans=ans+dp[3][0]+dp[3][1]+dp[3][2],得到 000～099、100～199、200～299 内符合条件的个数。

(2) 处理 2 位数,ans=ans+dp[2][0]+dp[2][1],得到 00～09、10～19 内符合条件的个数。实际上,这一步的计算对应的是 300～309、310～319。

(3) 处理 1 位数,即 ans=ans+dp[1][0]+dp[1][1]+dp[1][2]+dp[1][3]。实际上,这一步计算的是 320、321、322、323、324。

下面用记忆化搜索方法重新实现上述思路。

2. 用记忆化搜索实现 hdu 2089 题

回顾记忆化搜索，其思路是在递归程序 dfs()中搜索所有可能的情况，遇到已经算过的记录在 dp[]中的结果就直接使用，不再重复计算。

例如求[0，324]内符合条件的数，记忆化搜索的过程如图 7.25 所示。其中画线部分是前面已经算过的，记录在 dp[]中，不用再递归和重算。

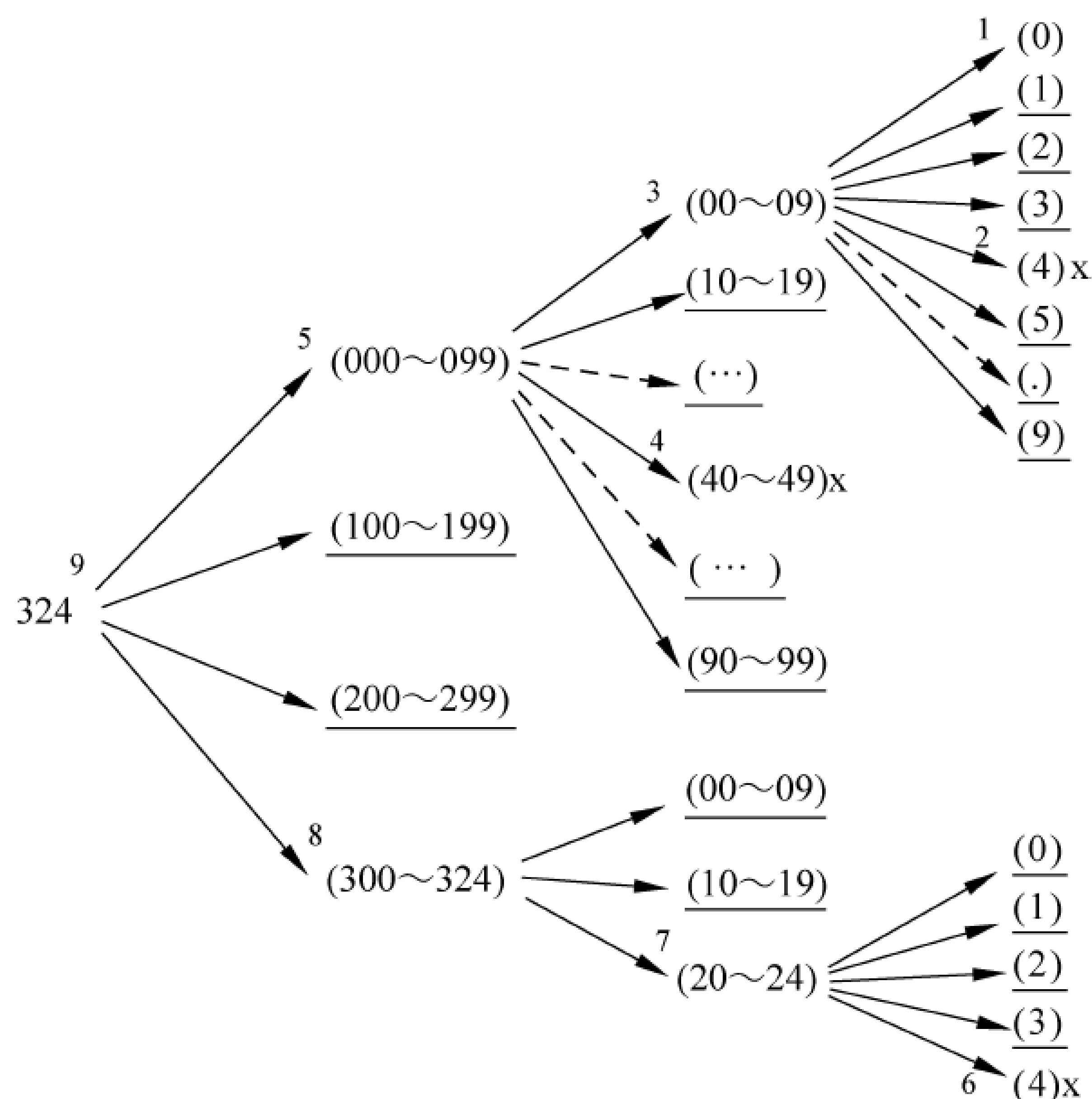


图 7.25　[0，324]的记忆化搜索过程

定义 dp[i]是 i 位数中符合要求的数字个数。dp[1]表示符合条件的 1 位数，0 是 1 位数，它的 dp[1]=1；1 也是 1 位数，它的 dp[1]沿用 0 算过的 dp[1]即可。dp[2]表示符合条件的 2 位数的个数，00～09 是 2 位数，计算得到 dp[2]=9；在搜索 10～19 时，沿用 dp[2]即可。同理，(100～199)和(200～299)都沿用(000～099)的计算结果 dp[3]，不用再计算。

dfs()的执行过程如下：从输入 324 开始，一直递归到最深处的(0)，然后逐步回退，计算的顺序是(0)→(4)→(00～09)→(40～49)→(000～099)→(4)→(20～24)→(300～324)→324，图 7.25 中用小写数字标识了这个顺序。

记忆化搜索极大地减少了搜索次数。例如图 7.25 中检查(000～099)，因为用 dp[]进行记忆化搜索，只需要计算 5 次；如果去掉记忆化部分，需要递归检查每个数，共 100 次。

下面是程序，程序只排除了数字'4'，读者自己练习排除"62"。对"62"的处理比较复杂，需要设计新的 dp[]状态。

```
#include <bits/stdc++.h>
const int LEN = 12;
int dp[LEN];            //dp[i]:i 位数符合要求的个数.例如 dp[2]表示 00～99 内符合要求的个数
int digit[LEN];
int dfs(int len, int ismax) {
    int ans = 0, maxx;
```

```
    if(!len) return 1;                  //已经递归到 0 位数,返回
    if(!ismax && dp[len]!= -1)          //记忆化搜索:如果已经算过,直接使用
        return dp[len];
    maxx = (ismax ? digit[len] : 9);
    for(int i = 0; i <= maxx; i++) {
        if(i == 4) continue;            //排除 4
        ans += dfs(len - 1, ismax && i == maxx);
    }
    if(!ismax)  dp[len] = ans;
    return ans;
}
int main(){
    int n, len = 0;
    memset(dp, -1, sizeof(dp));         //初始化为 -1
    scanf("%d", &n);
    while(n) {
        digit[++len] = n%10;
        n/= 10;
    }
    printf("%d\n", dfs(len,1));
    return 0;
}
```

【习题】

hdu 3555 题：求[1,N]里面有多少数包含'49'，$1\leqslant N\leqslant 2^{63}-1$。

hdu 3652 题：B-number 是一个非负整数，其十进制形式包含'13'并且可以被 13 整除。给定整数 n，$1\leqslant n\leqslant 10^9$，计算 $1\sim n$ 的 B-number 数。

hdu 6148 题：计算不大于 N 的 Valley Number 个数，结果对 10^9+7 取模。此题较难。

hdu 4507 题：计算[L,R]中和 7 无关的数字的平方和，结果对 10^9+7 取模。$1\leqslant L\leqslant R\leqslant 10^{18}$。此题较难。

7.6 状态压缩 DP

先用一道例题引出状态压缩 DP 的概念。

1. 例题 1

poj 3254 “Corn Fields”

农夫约翰有一片长方形土地，划成 M 行 N 列的方格。他准备种玉米、养牛，不过有些格子很贫瘠，不适合种玉米。还有，牛不喜欢在一起吃，所以牛不能放在相邻的格子里。给出这块地的情况，求约翰有多少个种玉米的方案。所有方格都不种玉米也算一种方案。



输入：第 1 行是 M 和 N，$1 \leq M, N \leq 12$。后面有 M 行，描述方格情况，1 表示肥沃，0 表示贫瘠。

输出：方案数，用 10^9 取模。

输入样例：

2 3

1 1 1

0 1 0

输出样例：

9

提示：在样例中有 9 种方案。

1	2	3
	4	

分别是{}、{1}、{2}、{3}、{4}、{1,3}、{1,4}、{3,4}、{1,3,4}。

这个方格图共 $m\times n$ 个格子，有 $2^{m\times n}$ 种排列，无法用暴力法计算。

用下面的方法编程计算，算法复杂度是 $O(m2^n2^n)$。

1）方格的表示

很容易想到，可以用二进制数来描述方格，1 表示种玉米，0 表示不种玉米。在样例中，第 1 行的 3 个方格都是肥沃的，排除相邻的情况，有以下 5 种种玉米的方案：

第 1 行	编　　号	1	2	3	4	5
	方　　案	000	001	010	100	101

这里的编号并不是多余的，在下面设计 DP 状态的时候有用。

2）DP 状态和状态转移

如何设计 DP 状态，把问题从小规模逐步扩展到大规模？可以按行进行扩展。

上面已经得到了第 1 行的 5 种方案，下面继续扩展第 2 行。

在样例中，第 2 行只有两种方案，即 000、010。

第 2 行	编　　号	1	2
	方　　案	000	010

如果第 2 行选编号 1 的 000，第 1 行可以选 5 种方案而不冲突。

如果第 2 行选编号 2 的 010，与第 1 行的 010 有冲突，第 1 行的其他 4 种方案没问题。

共 5+4=9 种方案。

用 dp[i][j]表示第 i 行采用第 j 种编号的方案时前 i 行可以得到的可行方案总数。例如，dp[2][2]=4 表示第 2 行使用第 2 种方案（即 010）时的方案总数是 4。

从第 $i-1$ 行转移到第 i 行，状态转移方程如下：

$$dp[i][k] = \sum_{j=1}^{n} dp[i-1][j]$$

其中 j 是第 $i-1$ 行可行方案的编号，而且所有的 $dp[i-1][j]$ 与第 i 行不冲突。

把最后一行的 $dp[m][k](1\leqslant k\leqslant n)$ 相加就得到了答案。

3）一些细节

程序有很多细节，例如初始化每一行的合法方案，即找没有相邻 1 的二进制数。用 state[]表示方案，例如样例的第 1 行 state[2]$=010_2$ 表示只种中间一块地。可以这样写程序：

```
int state[600];                          //state[x]:编号 x 的方案是 state[x]
bool check(int x){                       //判断 x 的二进制数是否有相邻的 1
    if(x&x << 1)return false;            //x 有相邻的 1,该方案不合法
    else      return true;               //x 没有相邻的 1,合法
}
void init(){                             //初始化合法的方案
    int j = 0;
    int total = 1 << N;                  //一行有 N 个格子,有 2^N 种情况
    for(int i = 0; i < total; ++i)
        if(check(i))   state[++j] = i;   //记录合法方案
}
```

对于相邻两行的合法性判断，这样写程序：

```
if(state[i] & state[j] != 1) …           //相邻的两行,没有挨着的 1
```

2. 状态压缩 DP 的概念

从上面的例子可以看出，每个状态 $dp[i][j]$ 表示的不是一个有意义的数值，例如前面章节中的花费、价值、长度等，而是代表了集合的数量。这种处理复杂集合问题的 DP 叫做状态压缩 DP。

集合的状态有很多，操作复杂，往往把方案用二进制数（“压缩”到这个二进制数中）来表示和操作。二进制操作有与、或、取反、移位等。在上面的例子中，把可能的方案“压缩”到 state[]中，操作用到了左移。

3. 旅行商问题


视频讲解

旅行商问题（Traveling Salesman Problem，TSP）是一个经典问题：有 n 个城市，已知任何两个城市之间的距离（或者费用），一个旅行商从某城市出发，经过每一个城市并且只经过一次，最后回到出发的城市，输出最短（或者路费最少）的线路。

TSP 问题是 NP 难度的，没有多项式时间的高效算法，所以 TSP 题目给的 n 都很小。如果用暴力法，可以列出所有的路线，然后逐一判断。路线最多可能有 $(n-1)!$ 种，只能用于解决规模 $n\leqslant 11$ 的问题。如果题目不需要求最短的线路，可以用贪心法求近似解，找出一条可行的、比较短的路线。

小规模的 TSP 问题可以用状态压缩 DP 求解，复杂度是 $O(2^n n^2)$，能解决规模 $n\leqslant 15$ 的问题，比暴力的 $O(n!)$ 好一些。思路如下：

假设最短的 TSP 路径是 Path$=(v_0 \to v_1 \to v_2 \to v_3 \to v_0)$

那么 Path$=(v_0\to v_1)+(v_1\to v_2\to v_3\to v_0)$

所以，问题转变为：求经过所有城市的最短回路→从某个城市到起点的最短路径。

DP 状态设计如下：假设已经访问过的城市集合是 S，当前所在城市是 v，用 dp$[S][v]$ 表示从 v 出发访问剩余的所有城市最后回到起点的路径费用总和的最小值。状态转移方程如下：

$$\mathrm{dp}[V][0]=0 \qquad //V \text{ 是最后一个城市}$$

$$\mathrm{dp}[S][v]=\min(\mathrm{dp}[S\cup\{u\}][u]+\mathrm{dist}(v,u)\mid u\notin S)$$

城市集合 S 如何表示？这就用到状态压缩 DP 的技巧：把路径"压缩"到一个二进制数中。定义：

```
int dp[1 << MAXN][MAXN];
```

MAXN 是城市数量，当 MAXN＝15 时，1 << MAXN $=2^{15}=32\ 768$，0～32 768 内的每个数的二进制表示就是一个可能的路径，二进制数中的 1 表示选中一个城市，0 表示不选中。例如 $S=000\ 0000\ 0000\ 0101_2$，末尾的 101 表示已访问过城市 2、0。在下面的代码中，"dp[s|1 << u][u]"，其中的 s|1 << u，表示在已访问过的城市集合 S 中加入一个新访问的城市 u。

下面是部分示意代码[①]。

```
int dp[1 << MAXN][MAXN];
void solve(){
    memset(dp, INF, sizeof(dp));                      //初始化为无穷大
    dp[(1 << n) - 1][0] = 0;                          //从最后一个点出发到起点 0,已经没有城市可
                                                      //以走,所以到起点 0 的最小路径费用是 0
    for (int s = (1 << n) - 2; s >= 0; s-- )          //O(2^n)
        for(int v = 0; v < n; v++)                    //O(n)
            for(int u = 0; u < n; u++)                //O(n)
                if(!(s >> u & 1))
                    dp[s][v] = min(dp[s][v], dp[s|1 << u][u] + dist[v][u]);
    printf("%d\n",dp[0][0]);
}
```

4. 例题 2

这一题是 TSP 的变形。

hdu 3001 "Travelling"

Acmer 先生决定访问 n 座城市。他可以空降到任意城市，然后开始访问，要求访问到所有城市，任何一个城市访问的次数不少于 1 次，不多于 2 次。n 座城市间有 m 条道路，每条道路都有路费。求 Acmer 先生完成旅行需要花费的最小费用。

输入：第 1 行是 n 和 m，$1\leqslant n\leqslant 10$；后面有 m 行，有 3 个整数 a、b、c，表示城市 a 和 b 之间的路费是 c。

输出：最少花费，如果不能完成旅行，则输出－1。

① 编程细节参考《挑战程序设计竞赛》(秋叶拓哉)192 页，3.4.1 节。

本题 $n=10$,数据很小,但是由于每个城市可以走两遍,可能的路线就变成了$(2n)!$,所以不能用暴力法。

本题用状态压缩 DP 求解,算法复杂度是 $O(3^n n^2)$,当 $n=10$ 时正好通过 OJ 测试。

1) 路径的表示

在普通 TSP 中,一个城市只有两种情况,即访问和不访问,用 1 和 0 表示。这个题有 3 种情况,也就是不访问、访问 1 次、访问 2 次,所以需要用到三进制。

当 $n=10$ 时有 3^{10} 种组合(路径数量),对每种路径用三进制表示。例如,第 14 种路径,它的三进制是 112_3,表示第 3 个城市走 1 次,第 2 个城市走 1 次,第 1 个城市走 2 次。

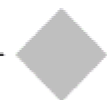
在程序中用 tri[i][j]表示第 i 个路径,其第 j 位的值是城市状态,例如 tri[14][3]=1, tri[14][2]=1, tri[14][1]=2。

2) 状态和状态转移

定义状态 dp[i][j],当前所在城市是 i,dp[i][j]表示从 i 出发访问剩余的所有城市最后回到起点的路径费用总和的最小值。

状态转移:dp[j][i]=min(dp[j][i], dp[k][l]+graph[k][j]);

```
#include <bits/stdc++.h>
const int INF = 0x3f3f3f3f;
using namespace std;
int n,m;
int bit[12] = {0,1,3,9,27,81,243,729,2187,6561,19683,59049};
                                        //三进制每一位的权值,与二进制的 0、1、2、4、8 等对照
int tri[60000][11];
int dp[11][60000];
int graph[11][11];                          //存图
void make_trb(){                            //初始化,求所有可能的路径
    for(int i = 0;i < 59050;++i){           //共 3^10 = 59 050 种状态
        int t = i;
        for(int j = 1; j <= 10; ++j){tri[i][j] = t % 3; t/ = 3;}
    }
}
int comp_dp(){
        int ans = INF;
        memset(dp, INF, sizeof(dp));
        for(int i = 0;i <= n;i++)
            dp[i][bit[i]] = 0;              //bit[i]是第 i 个城市,起点是任意的
        for(int i = 0;i < bit[n + 1];i++){
            int flag = 1;                   //所有的城市都遍历过 1 次以上
            for(int j = 1;j <= n;j++){      //选一个终点
                if(tri[i][j] ==  0){        //判断终点位是否为 0
                    flag = 0;               //还没有经过所有点
                    continue;
                }
                if(i == j) continue;
                for(int k = 1; k <= n; k++){
                    int l = i - bit[j];     //i 状态的第 j 位置 0
                    if(tri[i][k] == 0) continue;
```



```
                    dp[j][i] = min(dp[j][i],dp[k][l] + graph[k][j]);
                }
            }
            if(flag)                                 //找最小费用
                for(int j = 1; j <= n; j++)
                    ans = min(ans,dp[j][i]);
        }
        return ans;
}
int main(){
    make_trb();
    while(cin >> n >> m){
        memset(graph,INF,sizeof(graph));
        while(m--){
            int a,b,c;
            cin >> a >> b >> c;
            if(c < graph[a][b])  graph[a][b] = graph[b][a] = c;
        }
        int ans = comp_dp();
        if(ans == INF) cout <<"-1"<< endl;
        else           cout << ans << endl;
    }
    return 0;
}
```

【习题】

hdu 1074 “Doing Homework”，入门题。

hdu 2167 “Pebbles”。

hdu 3182 “Hamburger Magi”。

hdu 4539 “排兵布阵”。

poj 1185 “炮兵阵地”，经典题。

poj 2411 “Mondriaan's Dream”，铺砖问题。

hdu 3681 “Prison Break”，TSP＋二分，难题。

7.7　小　　结

本章介绍了常见的 DP 算法。读者已经看到，DP 题目不仅涉及大量知识点，而且思维灵活，不容易掌握。DP 题目作为竞赛的必考题型，参赛者需要花大量时间练习，掌握其中的诀窍。

另外还有很多可用 DP 的算法在本章没有涉及，例如用 DP 解决以概率为最优解的问题，具体内容见本书 8.4 节；还有 AC 自动机＋DP、后缀自动机＋DP 等。

第8章　数　　学

✍高精度计算
✍数论
✍组合数学
✍概率和数学期望
✍公平组合游戏

数学题在算法竞赛中经常出现。数学题的知识点相当广，有些容易理解，有些比较难。在竞赛中经常把数学模型和其他算法结合起来，出综合性的题目，所以本书把数学题相关内容放在比较靠后的章节。

常见的数学方面的题目包括数论、组合数学、概率和数学期望、组合游戏等大类，这里先列出常见的知识点，本章将讲解其中部分内容。

(1) 数论。

整除性问题：整除、最大公约数、最大公倍数；欧几里得算法、扩展欧几里得算法。

素数问题：素数判定、区间素数统计。

同余问题：模运算、同余方程、快速幂、中国剩余定理、逆元、整数分解、同余定理。

不定方程。

乘性函数：欧拉函数、伪随机数、莫比乌斯反演。

(2) 组合数学。

排列组合：计数原理、特殊排列、排列生成、组合生成。

母函数：普通型、指数型。

递推关系：Fibonacci 数列、Stirling 数、Catalan 数。

容斥原理、鸽巢原理。

群：Polya 定理。

线性规划：单纯形法。

(3) 矩阵、线性代数、高精度计算、概率和数学期望、组合游戏、傅里叶变换。

8.1　高精度计算



视频讲解

高精度计算，是指参与运算的数大大超出了标准数据类型所能表示的范围的运算，例如两个1000位数相乘。这类题目在算法竞赛中的出现很频繁。

在C或者C++中，最大的数据类型只有64位，如果需要处理更大的数，只能用数组来模

拟，把大数的每一位存储在数组中，然后按位处理进位、借位问题，相当麻烦。

但是用 Java 处理高精度非常简单，可以直接计算。在 Java 中有两个类——BigInteger 和 BigDecimal，分别表示大整数类和大浮点数类，两个类的对象能处理的数理论上能够表示无限大，只要计算机内存足够大。这两个类都在 java.math.* 包中。

例如 hdu 1042 题，输入整数 $N(0\leqslant N\leqslant 10\ 000)$，输出 $N!$。当 $N=10\ 000$ 时，$N!$ 是一个超级大的数字。读者可以尝试用 C++实现①。用 Java 可以直接算，下面是代码。

hdu 1042 题的 Java 代码

```
import java.math.BigInteger;
import java.util.*;
public class Main{
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        while(input.hasNext()) {
            int n = input.nextInt();
            BigInteger ans = BigInteger.ONE;
            for(int i = 1; i<=n; i++)
                ans = ans.multiply(BigInteger.valueOf(i));
            System.out.println(ans);
        }
    }
}
```

Java 虽然能处理大数，但是对于规模过大的问题用 Java 也不能做。例如 hdu 1061 题，$n=10^9$，求 n^n。此时需要一些特殊的算法，例如“快速幂”，见下一节内容。

【习题】

请读者自己找资料熟悉 Java 的高精度运算，并通过以下习题掌握用法。

hdu 1047，求和。

hdu 1063，实数的高精度幂。

hdu 1316，大数比较。

hdu 5666，大数除法。

hdu 5686，大数递推。

8.2　数　　论

数论是研究整数性质的数学分支。初等数论的主要内容有整除问题、素数、不定方程、同余问题、乘性函数等。本节介绍竞赛中常用的一些初等数论知识。

① 用“万进制”可以求解 $n!$，请读者搜索网上资料。

8.2.1 模运算

模运算是大数运算中的常用操作。如果一个数太大,无法直接输出,或者不需要直接输出,可以把它取模后缩小数值再输出。

定义取模运算为 a 除以 m 的余数[①],记为:

$$a \bmod m = a \ \% \ m$$

取模的结果满足 $0 \leqslant a \bmod m \leqslant m-1$,题目用给定的 m 限制计算结果的范围。例如 $m=10$,就是取计算结果的个位数,参考 hdu 1061 题,求 n^n,$n \leqslant 10^9$,输出结果的个位数。

取模操作满足以下性质。

加:$(a+b) \bmod m = ((a \bmod m)+(b \bmod m)) \bmod m$

减:$(a-b) \bmod m = ((a \bmod m)-(b \bmod m)) \bmod m$

乘:$(a \times b) \bmod m = ((a \bmod m) \times (b \bmod m)) \bmod m$

然而,对除法取模进行下面的类似操作是错误的:

$$(a/b) \bmod m = ((a \bmod m)/(b \bmod m)) \bmod m$$

例如,$(100/50) \bmod 20 = 2$,$(100 \bmod 20)/(50 \bmod 20) \bmod 20 = 0$,两者不相等。

除法的取模需要用到逆元,将在“同余与逆元”这一节中介绍。

8.2.2 快速幂

1. 快速幂概念

快速幂以及扩展的矩阵快速幂,由于应用场景比较常见,也是竞赛中常见的题型。

幂运算 a^n 即 n 个 a 相乘。快速幂就是高效地算出 a^n。当 n 很大时,例如 $n=10^9$,计算 a^n 这样大的数 Java 也不能处理,一是数字太大,二是计算时间很长。下面先考虑如何缩短计算时间,如果用暴力的方法直接算 a^n,即逐个做乘法,复杂度是 $O(n)$,即使能算出来,也会超时。

读者很容易想到快速幂的办法:先算 a^2,然后继续算平方 $(a^2)^2$,一直算到 n 次幂。这是分治法的思想,复杂度为 $O(\log_2 n)$。下面是代码,请读者自己理解:

```
int fastPow(int a, int n){
   if(n == 1)   return a;
   int temp = fastPow(a, n/2);          //分治
   if(n%2 == 1)                         //奇数个 a,此处也可以写为 if(n &1)
        return temp * temp * a;
   else                                 //偶数个 a
        return temp * temp;
}
```

程序中的递归,层数只有 $\log_2 n$,不用担心溢出的问题。

上面的程序非常好,不过还有一种更好的方法,是用位运算做快速幂,时间复杂度也是 $O(\log_2 n)$。下面以 a^{11} 为例说明快速幂的原理。

---

① 注意,此时要求 a 和 m 的正负号一致,都为正数或都为负数。如果正负不同,取模和求余的结果是不同的。



先把 a^{11} 分解成 a^8、a^2、a^1 的乘积，即 $a^{11}=a^{8+2+1}=a^8\times a^2\times a^1$。

如何求 a^8、a^2、a^1 的值，需要分别计算吗？并不需要。用户可以容易地发现，$a^1\times a^1=a^2$，$a^2\times a^2=a^4$，$a^4\times a^4=a^8$，等等，都是2的倍数，产生的 a^i 都是倍乘关系，逐级递推就可以了。在下面的程序中，这个功能用"base *=base;"实现。

那么如何把 n 分解成 $11=8+2+1$ 这样的倍乘关系？用二进制就能理解了。把 n 转为二进制数，二进制数中每一位的权值都是低一位的两倍，对应的 a^i 是倍乘的关系，例如 $n=11_{10}=1011_2=2^3+2^1+2^0=8+2+1$，所以只需要把 n 按二进制处理就可以了。

另外还有一个需要处理的问题：如何跳过那些不需要的？例如求 a^{11}，因为 $11=8+2+1$，需要跳过 a^4。这里做个判断即可，1011 中的 0 就是需要跳过的。这个判断，利用二进制的位运算很容易实现：

(1) $n \,\&\, 1$，取 n 的最后一位，并且判断这一位是否需要跳过。

(2) $n>>=1$，把 n 右移一位，目的是把刚处理过的 n 的最后一位去掉。

```
int fastPow(int a, int n){
    int base = a;                 //不定义 base,直接用 a 进行计算也行
    int res = 1;                  //用 res 返回结果
    while(n) {
        if(n & 1)                 //如果 n 的最后一位是 1,表示这个地方需要乘
            res *= base;
        base *= base;             //推算乘积,a² --> a⁴ --> a⁸ --> a¹⁶ …
        n >>= 1;                  //n 右移一位,把刚处理过的 n 的最后一位去掉
    }
    return res;
}
```

对照上面的程序，执行步骤如表 8.1 所示。

表 8.1　执行步骤

	n	res(res *=base)	base(base *=base)
第 1 轮	1011	a^1	a^2
第 2 轮	101	$a^1\times a^2$	a^4
第 3 轮	10	是 0，res 不变	a^8
第 4 轮	1	$a^1\times a^2\times a^8$	a^{16}
结束	0		

2. 快速幂取模

由于幂运算的结果非常大，常常会超过变量类型的最大值，甚至超过内存所能存放的最大数，所以涉及快速幂的题目，通常都要做取模操作，缩小结果。

根据模运算的性质，在快速幂中做取模操作，对 a^n 取模，和先对 a 取模再做幂运算的结果是一样的，即：

$$a^n \bmod m=(a \bmod m)^n \bmod m$$

下面修改位运算 fastPow()函数，加上取模操作。以 hdu 2817 题为例，取模操作如下：

```
const int mod = 200907;
…
```

```
if(n & 1)
    res = (base * res) % mod;
base = (base * base) % mod;
...
```

对于分治法 fastPow()函数的取模操作,请读者自己做类似的修改。

3. 矩阵快速幂

给定一个 $m\times m$ 的矩阵 $\boldsymbol{A}$,求它的 n 次幂 $\boldsymbol{A}^n$,这也是常见的计算。同样有矩阵快速幂的算法,原理是把矩阵当作变量来操作,程序和上面的很相似。

首先需要定义矩阵的结构体,并且定义矩阵相乘的操作。注意矩阵相乘也需要取模。

```
const int MAXN = 2;                        //根据题目要求定义矩阵的阶,本例中是 2
const int MOD = 1e4;                       //根据题目要求定义模
struct Matrix{                             //定义矩阵
    int m[MAXN][MAXN];
    Matrix() {
        memset(m, 0, sizeof(m));
    }
};
Matrix Multi(Matrix a, Matrix b) {         //矩阵的乘法
    Matrix res;
    for(int i = 0; i < MAXN; i++)
        for(int j = 0; j < MAXN; j++)
            for(int k = 0; k < MAXN; k++)
                res.m[i][j] = (res.m[i][j] + a.m[i][k] * b.m[k][j]) % MOD;
    return res;
}
```

下面是矩阵快速幂的程序代码,和前面单变量的快速幂的代码非常相似。

```
Matrix fastm(Matrix a, int n){
    Matrix res;
    for(int i = 0; i < MAXN; i++)
            //初始化为单位矩阵,相当于前面程序中的"int res = 1;"
      res.m[i][i] = 1;
    while(n) {
        if(n&1)
            res = Multi(res, a);
        a = Multi(a, a);
        n >>= 1;
    }
    return res;
}
```

矩阵快速幂的复杂度:上面求 $\boldsymbol{A}^n$,$\boldsymbol{A}$ 是 $m\times m$ 的方阵,其中矩阵乘法的复杂度是 $O(m^3)$,快速幂的复杂度是 $O(\log_2 n)$,合起来是 $O(m^3\log_2 n)$。

应用矩阵快速幂的难点在于如何把递推关系转换为矩阵。

【习题】

hdu 2817。

hdu 1061，求 n^n 的末尾数字，$n \leqslant 10^9$。

hdu 5392，快速幂取模、LCM。

poj 3070、hdu 3117。矩阵快速幂的经典题目，算 Fibonacci 数列。求第 10^9 个 Fibonacci 数，因为直接递推无法完成，所以先用矩阵表示 Fibonacci 数列的递推关系，然后把问题转换为求这个矩阵的 10^9 幂。

hdu 6030，把递推关系转换为矩阵。

hdu 5895，有难度的矩阵快速幂。

hdu 5564，数位 DP、矩阵快速幂。

hdu 2243，AC 自动机、矩阵快速幂。

8.2.3　GCD、LCM

最大公约数 GCD 和最小公倍数 LCM 是竞赛中常见的知识点，虽然这两个知识点很容易理解，但往往会与其他知识点结合起来出综合题，并不容易。

1. 最大公约数 GCD

整数 a 和 b 的最大公约数记为 $\gcd(a,b)$。在编程时有两种做法。

(1) 经典的欧几里得算法，用辗转相除法求最大公约数，模板如下：

```
int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a % b);
}
```

时间复杂度差不多是 $O(\log_2 n)$ 的[①]，非常快。

(2) 或者直接用 C++的内置函数求 GCD：

```
std::__gcd(a, b)
```

2. 最小公倍数 LCM

整数 a 和 b 的最小公倍数记为 $\text{lcm}(a,b)$，模板如下：

```
int lcm(int a, int b) {
    return a/gcd(a, b) * b;
}
```

8.2.4　扩展欧几里得算法与二元一次方程的整数解

读者可能还记得中学接触过的一个问题：给出整数 a、b、n，问方程 $ax+by=n$ 什么时

① 严格的复杂度分析参考《初等数论及其应用》第 6 版，Kenneth H. Rosen 著，夏鸿刚译，机械出版社，3.4 节的欧几里得算法。

候有整数解？如何求所有的整数解？

有解的充分必要条件是 $\gcd(a,b)$ 可以整除 n。简单解释如下：

令 $a=\gcd(a,b)a'$、$b=\gcd(a,b)b'$，有 $ax+by=\gcd(a,b)(a'x+b'y)=n$；如果 x、y、a'、b' 都是整数，那么 n 必须是 $\gcd(a,b)$ 的倍数才有整数解。

例如 $4x+6y=8$、$2x+3y=4$ 有整数解，$4x+6y=7$ 则没有整数解。

如果确定有解，一种解题方法是先找到一个解 (x_0,y_0)，那么通解公式如下：

$$x = x_0 + bt$$
$$y = y_0 - at, \quad t\text{ 是任意整数}$$

所以，问题转化为如何求 (x_0,y_0)。利用扩展欧几里得算法可以求出这个特解。

1. 扩展欧几里得算法

当方程符合 $ax+by=\gcd(a,b)$ 时，可以用扩展欧几里得算法求 (x_0,y_0)。程序如下[①]：

```
void extend_gcd(int a, int b, int &x, int &y){        //返回 x,y,即一个特解(x₀,y₀)
   if(b == 0) {
      x = 1, y = 0;
      return;
   }
   extend_gcd(b, a % b, x, y);
   int tmp = x;
   x = y;
   y = tmp - (a/b) * y;
}
```

有时候为了简化描述，在 $ax+by=\gcd(a,b)$ 两边除以 $\gcd(a,b)$，得到 $cx+dy=1$，其中 $c=a/\gcd(c,b)$，$d=b/\gcd(a,b)$。很明显，c、d 是互质的。$cx+dy=1$ 的通解如下：

$$x = x_0 + dt$$
$$y = y_0 - ct, \quad t\text{ 是任意整数}$$

2. 求任意方程 $ax+by=n$ 的一个整数解

用扩展欧几里得算法求解 $ax+by=\gcd(a,b)$ 后，利用它可以进一步解任意方程 $ax+by=n$，得到一个整数解。其步骤如下：

(1) 判断方程 $ax+by=n$ 是否有整数解，有解的条件是 $\gcd(a,b)$ 可以整除 n；

(2) 用扩展欧几里得算法求 $ax+by=\gcd(a,b)$ 的一个解 (x_0,y_0)；

(3) 在 $ax_0+by_0=\gcd(a,b)$ 两边同时乘以 $n/\gcd(a,b)$，得：

$$ax_0n/\gcd(a,b) + by_0n/\gcd(a,b) = n$$

(4) 对照 $ax+by=n$，得到它的一个解 (x_0',y_0') 是：

$$x_0' = x_0n/\gcd(a,b)$$
$$y_0' = y_0n/\gcd(a,b)$$

3. 应用场合

扩展欧几里得算法是一个很有用的工具，在竞赛题目中常用于以下场合：

① 程序的执行过程参考《算法导论》，Thomas H. Cormen 等著，机械工业出版社，31.2 节。

(1) 求解不定方程；

(2) 求解模的逆元；

(3) 求解同余方程。

虽然用扩展欧几里得算法可以算 $ax+by=\gcd(a,b)$ 的通解，不过一般没有这个需求，而是用于求某些特殊的解，例如求解逆元，逆元是除法取模操作常用的工具。

【习题】

poj 1061，扩展欧几里得。
hdu 1019，LCM。
hdu 1576，扩展欧几里得。
hdu 2504，GCD，水题。
hdu 2588，GCD，欧拉函数。
hdu 5223，GCD，贪心。
hdu 5584，LCM。
hdu 5656，GCD，DP。
hdu 5902，GCD，暴力。

8.2.5 同余与逆元

同余在数论中非常有用，它用类似处理等式的方式来处理整除关系，非常简便。

1. 同余

两个整数 a、b 和一个正整数 m，如果 a 除以 m 所得的余数和 b 除以 m 所得的余数相等，即 $a \bmod m=b \bmod m$，称 a 和 b 对于 m 同余[①]，m 称为同余的模。同余的概念也可以这样理解：$m|(a-b)$，即 $a-b$ 是 m 的整数倍。例如 $6|(23-5)$，23 和 5 对模 6 同余。

同余的符号记为 $a\equiv b \pmod m$。

2. 一元线性同余方程

$ax\equiv b \pmod m$，即 ax 除以 m，b 除以 m，两者余数相同，这里 a、b、m 都是整数，求解 x 的值。

方程也可以这样理解：$ax-b$ 是 m 的整数倍。设 y 是倍数，那么 $ax-b=my$，移项得到 $ax-my=b$。因为 y 可以是负数，改写为 $ax+my=b$，这就是在扩展欧几里得算法中提到的二元一次不定方程。

当且仅当 $\gcd(a,m)$ 能整除 b 时有整数解。例如 $15x+6y=9$，有整数解 $x=1, y=-1$。

当 $\gcd(a,m)=b$ 时，可以直接用扩展欧几里得算法求解 $ax+my=b$。

如果不满足 $\gcd(a,m)=b$，还能用扩展欧几里得算法求解 $ax+my=b$ 吗？答案是肯定的，但是需要结合下面的逆元。

3. 逆元

给出 a 和 m，求解方程 $ax\equiv 1 \pmod m$，即 ax 除以 m 余数是 1。

① 《初等数论及其应用》第 6 版，Kenneth H. Rosen 著，夏鸿刚译，机械工业出版社，第 4 章“同余”。

根据前面的讨论,有解的条件是 $\gcd(a,m)=1$,即 a、m 互素。该问题等价于求解 $ax+my=1$,可以用上一节的扩展欧几里得算法求解。例如 $8x\equiv1(\bmod 31)$,等价于求解 $8x+31y=1$,用扩展欧几里得算法求得一个特解是 $x=4$,$y=-1$。

方程 $ax\equiv1(\bmod m)$的一个解 x,称 x 为 a 模 m 的逆。注意,这样的 x 有很多,把它们都称为逆。

求逆元的代码如下:

```
int mod_inverse(int a, int m){
    int x, y;
    extend_gcd(a, m, x, y);
    return(m + x % m) % m;            //x 可能是负数,需要处理
}
```

另外,在某些情况下也可以用费马小定理求逆元。

4. 逆元与除法取模

逆元的一个重要应用是求除法的模。在后面讲 Catalan 数的时候有这样一个需求:求 $(a/b)\bmod m$,即 a 除以 b,然后对 m 取模。由于这里 a 和 b 都是很大的数,做除法后再取模会损失精度。下面的方法可以避开除法计算。

设 b 的逆元是 k,有:

$$\left(\frac{a}{b}\right)\bmod m=\left(\left(\frac{a}{b}\right)\bmod m\right)((bk)\bmod m)=\left(\frac{a}{b}bk\right)\bmod m=(ak)\bmod m$$

上述推导过程把除法的模运算转换成了乘法模运算:$(a/b)\bmod m=(ak)\bmod m$

5. 逆元与求解二元一次方程 $ax+my=b$

如果得到了 a 的逆,可以来求解形如 $ax\equiv b(\bmod m)$的任何同余方程。方法如下:令 a'是 a 模 m 的逆,则 $a'a\equiv1(\bmod m)$;在 $ax\equiv b(\bmod m)$的两边同时乘以 a',得到 $a'ax\equiv a'b(\bmod m)$,所以 $x\equiv a'b(\bmod m)$。

例如求 $8x\equiv24(\bmod 31)$的解。先求 8 模 31 的逆,是 4;然后在 $8x\equiv24(\bmod 31)$的两边乘以 4,得到 $8\times4x\equiv4\times24(\bmod 31)$,所以 $x\equiv96(\bmod 31)$。

前面讲解扩展欧几里得算法时曾求解了二元一次方程,这里再给出利用逆元的另一种方法,如表 8.2 所示。读者对照两种方法,可以加深对逆元的理解。

表 8.2 利用逆元的求解方法

步骤	求解方程 $ax+my=b$ 同余方程是 $ax\equiv b(\bmod m)$	例:求解 $8x+31y=24$ 同余方程是 $8x\equiv24(\bmod 31)$ $a=8,b=24,m=31$
1	有解的条件:$\gcd(a,m)$能整除 b	$\gcd(8,31)=1$ 能整除 24
2	求 $ax\equiv1(\bmod m)$的逆元 a',等价于用扩展欧几里得算法求解 $ax+my=1$	$8x+31y=1$ 的一个解是 $x=4,y=-1$ 即一个逆元是 $a'=4$
3	一个特解是 $x=a'b$	$x=a'b=4\times24=96$
4	代入方程 $ax+my=b$,求解 y	代入 $8x+31y=24$,得到 $y=-24$

【习题】

hdu 5976 "Detachment",乘法逆元。

8.2.6 素数

1. 用试除法判断素数

问题:输入一个很大的数 n,判断它是不是素数。

素数定义:一个数 n,如果不能被$[2,n-1]$内的所有数整除,n 就是素数。当然,并不需要把$[2,n-1]$内的数都试一遍,这个范围可以缩小到$[2,\sqrt{n}]$。

给定 n,如果它不能整除$[2,\sqrt{n}]$内的所有数,它就是素数。证明如下:

设 $n=a\times b$,有 $\min(a,b)\leqslant\sqrt{n}$,令 $a\leqslant b$。只要检查$[2,\sqrt{n}]$内的数,如果 n 不是素数,就能找到一个 a。如果不存在这个 a,那么$(\sqrt{n},n-1]$内也不存在 b。

以上判断的范围可以再缩小一点:$[2,\sqrt{n}]$内所有的**素数**。其原理很简单,读者在学过下文提到的埃式筛法之后更容易理解。

用试除法判断素数,复杂度是 $O(\sqrt{n})$,对于 $n\leqslant10^{12}$ 的数是没有问题的。

下面是试除法判断素数的代码。

判断素数

```
bool is_prime(int n){
    if(n <= 1)  return false;            //1 不是素数
    for(int i = 2; i * i <= n; i++)      //比这样写更好: for(int i = 2;i <= sqrt(n);i++)
        if(n % i == 0)  return false;    //能整除,不是素数
    return true;
}
```

2. 巨大素数的判断

如果 n 非常大,例如 poj 1811 题,$1\leqslant n<2^{54}$,判断 n 是不是素数。如果用试除法,$\sqrt{n}=2^{27}\approx10^8$,复杂度仍然太高。此时需要用到特殊而复杂的方法,如果读者有兴趣,可以自己查资料①。

3. 用埃式筛法求素数的数量

一个与素数相关的问题是求$[2,n]$内所有的素数。如果用上面的试除法,一个个单独进行判断,太慢了。

埃式筛法是一种古老而简单的方法,可以快速找到$[2,n]$内所有的素数。对于初始队列$\{2,3,4,5,6,7,8,9,10,11,12,13,\cdots,n\}$,操作步骤如下:

(1) 输出最小的素数 2,然后筛掉 2 的倍数,剩下$\{3,5,7,9,11,13,\cdots\}$;

① 《ACM/ICPC 算法训练教程》,余立功,清华大学出版社,127 页。

(2) 输出最小的素数 3，然后筛掉 3 的倍数，剩下{5,7,11,13,…}；

(3) 输出最小的素数 5，然后筛掉 5 的倍数，剩下{7,11,13,…}。

继续以上步骤，直到队列为空。

下面是程序，其中 visit[i]记录数 i 的状态，如果 visit[i]=true，表示它被筛掉了，不是素数。用 prime[]存放素数，例如 prime[0]是第 1 个素数 2。

```
const int MAXN = 1e7;                       //定义空间大小,1e7 约 10MB
int prime[MAXN + 1];                        //存放素数,它记录 visit[i] = false 的项
bool visit[MAXN + 1];                       //true 表示被筛掉,不是素数
int E_sieve(int n)  {                       //埃式筛法,计算[2, n]内的素数
    int k = 0;                              //统计素数的个数
    for(int i = 0; i <= n; i++)  visit[i] = false;     //初始化
    for(int i = 2; i <= n; i++) {           //从第 1 个素数 2 开始.可优化(1)
        if(!visit[i]) {
            prime[k++] = i;                 //i 是素数,存储到 prime[]中
            for(int j = 2 * i; j <= n; j += i)   //i 的倍数都不是素数。可优化(2)
                visit[j] = true;            //标记为非素数,筛掉
        }
    }
    return k;                               //返回素数的个数
}
```

计算复杂度：2 的倍数被筛掉，计算 $n/2$ 次；3 的倍数被筛掉，计算 $n/3$ 次；5 的倍数被筛掉，计算 $n/5$ 次，依此类推。总次数是 $O(n/2+n/3+n/5+\cdots)$，这里直接给出结果，即 $O(n\log\log_2 n)$。

空间复杂度：程序用到了 bool visit[MAXN+1]数组，当 MAXN=10^7 时约 10MB。一般题目会限制空间为 65MB，所以 n 不能再大了。

上述代码有两处可以优化：

(1) 用来做筛除的数为 2、3、5 等，最多到$\sqrt{n}$就可以了。例如求 $n=100$ 以内的素数，用 2、3、5、7 筛就足够了。其原理和试除法一样：非素数 k 必定可以被一个小于等于$\sqrt{k}$的素数整除。

(2) for(int j=2*i; j<=n; j+=i)中的 $j=2*i$ 优化为 $j=i*i$。例如 $i=5$ 时，2*5、3*5、4*5 已经在前面 $i=2,3,4$ 的时候筛过了。

优化后的代码如下：

```
int E_sieve(int n) {
    for(int i = 0; i <= n; i++)  visit[i] = false;
    for(int i = 2; i * i <= n; i++)         //筛掉非素数
        if(!visit[i])
            for(int j = i * i; j <= n; j += i)
                visit[j] = true;            //标记为非素数
        //下面记录素数
    int  k = 0;                             //统计素数的个数
    for(int i = 2; i <= n; i++)
        if(!visit[i])
```

```
        prime[k++] = i;                  //存储素数
    return k;
}
```

埃式筛法虽然还不错，但其实做了一些无用功，某个数会被筛几次，比如12，被2和3筛了两次。另一种欧拉筛选法，时间复杂度仅为$O(n)$，如果读者有兴趣可以查资料。不过，埃式筛法可以近似看成$O(n)$的，一般也够用了。

4. 埃式筛法应用于大区间素数

用埃式筛法求$[2,n]$内的素数，只能解决规模$n \leqslant 10^7$的问题。如果n更大，在某些情况下可以用埃式筛法来处理，这就是大区间素数的计算。

如果把$[2,n]$看成一个区间，那么可以把埃式筛法扩展到求区间$[a,b]$的素数，$a<b\leqslant 10^{12}$，$b-a\leqslant 10^6$。

前文提到，用试除法判断n是不是素数，原理为：如果它不能整除$2\sim\sqrt{n}$内所有的素数，它就是素数。根据埃式筛法很容易理解这个原理：$2\sim\sqrt{n}$内的非素数b肯定对应一个比它小的素数a。在用试除法的时候，如果n能整除a，已经证明了n不是素数，b就不用再试了。

这个原理可以用来理解大区间求素数问题。先用埃式筛法求$[2,\sqrt{b}]$内的素数，然后用这些素数来筛$[a,b]$区间的素数即可。

(1) 计算复杂度：$O(\sqrt{b}\log\log\sqrt{b})+O((b-a)\sqrt{b-a})$；

(2) 空间复杂度：需要定义两个数组，一个用于处理$[2,\sqrt{b}]$内的素数，另一个用于处理$[a,b]$内的素数，空间复杂度是$O(\sqrt{b})+O(b-a)$。

习题：poj 2689，求$[L,R]$内的素数，$1\leqslant L<R\leqslant 2\,147\,483\,647$)，$R-L\leqslant 10^6$。

5. 更大的素数

上面埃式筛法的限制条件是$n\leqslant 10^7$。如果要统计更大范围内的素数个数，例如$n=10^{11}$时有40多亿个素数①，此时需要用到更复杂的数学方法。如果读者有兴趣可以研究hdu 5901 Count primes一题，求$1\leqslant n\leqslant 10^{11}$范围内的素数个数。

【习题】

hdu 1262，寻找素数对。
hdu 2710，筛法求素数。
hdu 3792，素数打表。
hdu 3826，分解质因子。
hdu 6069，区间素数。

---

① $[2,n]$内素数的数量：https://en.wikipedia.org/wiki/Prime-counting_function（永久网址：perma.cc/MSN6-F4AM）。

8.3 组合数学

人们在生活中经常会遇到排列组合问题。简单的，例如在5个礼物中选两个，问有多少种选取方法？复杂一点的，例如一串手环，用不同颜色的珠子串成，问有多少种不同的排列方法？

组合数学就是研究一个集合内满足一定规则的排列问题。这类问题如下：

(1) 存在问题，即判断这些排列是否存在；

(2) 计数问题，计算出有多少种排列，并构造出来；

(3) 优化问题，如果有最优解，给出最优解。

组合数学涉及的内容很多，包括①：

(1) 基本计数规则，例如乘法规则、加法规则、生成排列组合、多项式系数、鸽巢(抽屉)原理等。

(2) 计数问题，例如母函数(普通型、指数型、概率型等)、二项式定理、递推关系、容斥定理、Pólya定理等。

(3) 存在问题，例如编码、组合设计、图论中的存在问题等。

(4) 组合优化，例如匹配和覆盖、图和网络的优化问题。

本书的内容涉及前两部分，即计数规则和计数问题。

8.3.1 鸽巢原理

鸽巢原理(Pigeonhole Principle)，或称抽屉原理(Drawer Principle)，内容非常简单：把$n+1$个物体放进n个盒子，至少有一个盒子包含两个或更多的物体。

鸽巢原理是很基本的组合原理，但是可以解决许多有趣的问题，得到一些有趣的结论。例如：在1500人中，至少有5人生日相同；n个人互相握手，一定有两个人握手的次数相同。

hdu 1205 "吃糖果"

Gardon有K种糖果，每种数量已知，Gardon不喜欢连续两次吃同样的糖果，问有没有可行的吃糖方案。

该题是非常典型的鸽巢原理问题，可以用"隔板法"求解。找出最多的一种糖果，把它的数量N看成N个隔板，隔成N个空间(把每个隔板的右边看成一个空间)；其他所有糖果的数量为S。

(1) 如果$S<N-1$，把S个糖果放到隔板之间，这N个隔板不够放，必然至少有两个隔

① 参考《应用组合数学》，Fred S. Roberts、Barry Tesman著，冯速译，机械工业出版社。这本书几乎包罗了所有的组合数学知识。这类书的特点是过于详细，读者不可能通读所有内容，也不太容易提炼出一些知识点的算法思想。建议读者边做竞赛题边查阅有关知识，这样能很快地把知识与应用结合起来。

板之间没有糖果，由于这两个隔板是同一种糖果，所以无解。

(2) 当 $S \geqslant N-1$ 时，肯定有解。其中一个解是把 S 个糖果排成一个长队，注意同种类的糖果是挨在一起的，然后每次取 N 个糖果，按顺序一个一个地放进 N 个空间。由于隔板的数量比每一种糖果的数量都多，所以不可能有两个同样的糖果被放进一个空间里。把 S 个糖果放完，就是一个解，一些隔板里面可能放几种糖果。

鸽巢原理是 Ramsey 定理的一个特例。读者可以通过这两题来了解 Ramsey 定理，即 hdu 5917/6152，它们是 2016、2017 年的比赛题。

【习题】

poj 2356/3370。

hdu 1808/3183/5776。

8.3.2 杨辉三角和二项式系数

读者一定非常熟悉排列和组合公式。

排列：$A_n^k = \dfrac{n!}{(n-k)!}$

组合：$C_n^k = \dbinom{n}{k} = \dfrac{A_n^k}{k!} = \dfrac{n!}{k!(n-k)!}$

这里把组合数 C_n^k 用符号 $\dbinom{n}{k}$ 表示，称为二项式系数(Binomial Coefficient)。

杨辉三角(国外称帕斯卡三角)是二项式系数 $\dbinom{n}{r}$ 的典型应用。

杨辉三角是排列成如下三角形的数字：

```
          1
        1   1
      1   2   1
    1   3   3   1
  1   4   6   4   1
1   5   10  10  5   1
```

每一行从上一行推导而来。如果编程求杨辉三角第 n 行的数字，可以模拟这个推导过程，逐级递推，复杂度是 $O(n^2)$。不过，若改用数学公式计算，则可以直接得到结果，比用递推快多了，这个公式就是 $(1+x)^n$。

观察 $(1+x)^n$ 的展开：

$$(1+x)^0 = 1$$
$$(1+x)^1 = 1+x$$
$$(1+x)^2 = 1+2x+x^2$$
$$(1+x)^3 = 1+3x+3x^2+x^3$$

每一行展开的系数刚好对应杨辉三角每一行的数字。也就是说，杨辉三角可以用

$(1+x)^n$ 来定义和计算。

那么如何计算 $(1+x)^n$？真的需要展开算系数吗？并不需要，二项式系数 $\binom{n}{k}=\frac{n!}{k!(n-k)!}$ 就是 $(1+x)^n$ 展开后的系数。它们的关系可以这样理解：$(1+x)^n$ 的第 k 项，实际上就是从 n 个 x 中选出 k 个，这就是组合数 $\binom{n}{k}$ 的定义。所以：

$$(1+x)^n=\sum_{k=1}^{n}\binom{n}{k}x^k$$

这个公式称为二项式定理。

有了这个公式，在求杨辉三角第 n 行的数字时就可以用公式直接计算了，复杂度为 $O(1)$。不过，该公式中有 $n!$，如果直接计算 $n!$，由于太大，有可能溢出。例如 hdu 2032 题，$n=30$，30! 超过了 long long 的范围。此时可以利用 $\binom{n}{k-1}$ 和 $\binom{n}{k}$ 的递推关系 $\frac{\binom{n}{k}}{\binom{n}{k-1}}=\frac{n-k+1}{k}$ 逐个推导，避免计算阶乘。

8.3.3 容斥原理

容斥原理(Inclusion-Exclusion Principle)是常见的思维方法。在计数时，有时情况比较多，相互有重叠。为了使重叠部分不被重复计算，可以这样处理：先不考虑重叠的情况，把所有对象的数目计算出来，然后减去重复计算的数目。这种计数方法称为容斥原理。

例如一根长为 60m 的绳子，每隔 3m 做一个记号，每隔 4m 也做一个记号，然后把有记号的地方剪断，问绳子共被剪成了多少段？容斥原理的解题思路是：3 的倍数有 20 个，不算绳子两头，有 20－1＝19 个记号；4 的倍数有 15 个；既是 3 的倍数又是 4 的倍数的，有 60÷(3×4)＝5 个。所以记号的总数量是(20－1)＋(15－1)－(5－1)＝29，绳子被剪成 29 段。

【习题】

hdu 2841/4135/4497/5155。

8.3.4 Fibonacci 数列

Fibonacci 数列是一个很常见的递推数列，在小学奥数中被称为“兔子数列”。Fibonacci 数列也是一个被“神话”的数列，人们常常提到的“黄金分割”就蕴含在 Fibonacci 数列中。

1. Fibonacci 数列的递推公式

$$f(1)=f(2)=1$$
$$f(n)=f(n-1)+f(n-2)$$

从第 3 项开始，每一项都等于前 2 项之和，前一部分数是 1,1,2,3,5,8,13…

当 n 趋于无穷大时，相邻两个数的比值 $f(n)/f(n-1)\rightarrow 0.618\,033\,988\,7\cdots$ 这就是有名的黄金分割数[①]。

2. 计算 Fibonacci 数列

这里有两个问题：①如何更快地计算第 n 个 Fibonacci 数；②Fibonacci 数增长太快了，需要处理大数。

那么如何计算 Fibonacci 数列？如果只计算到第 10^6 个数，用上面的递推公式就可以，复杂度是 $O(n)$。如果更大，例如算第 1 亿个数，这样就比较慢。对于这么大的 Fibonacci 数，需要用一种巧妙的方法：把递推关系转换成矩阵，并用前面讲过的矩阵快速幂进行处理。请读者自己做 poj 3070 题和 hdu 3117 题，求第 1 亿个 Fibonacci 数，这是一个**必做题**。

另外，Fibonacci 数的值增长非常快，近似于 $O(2^n)$，例如第 40 个数是 102 334 155，已经非常大，所以常常需要处理大数，或者做取模操作。由于这些知识在前面讲过，这里不再赘述。

3. 应用模型

Fibonacci 数列看起来很简单，但是应用却非常广泛。例如在排列组合问题中，很多场景的数学模型就是 Fibonacci 数列。下面是两个常见的例子。

1）楼梯问题

hdu 2041 题。有一楼梯共 M 级，刚开始时人在第一级，若每次只能跨上一级或两级，要走上第 M 级，共有多少种走法？

假设到第 n 级总共的走法为 $f(n)$。如何到达第 n 级？可以分成两种情况：①第一次跳 1 级，剩下 $n-1$ 个台阶，跳法是 $f(n-1)$；②第一次跳 2 级，剩下 $n-2$ 个台阶，跳法是 $f(n-2)$，所以 $f(n)=f(n-1)+f(n-2)$。这是一个 Fibonacci 数列。

2）矩形覆盖问题

用 2×1 的小矩形覆盖 $2n$ 的大矩形，总共有多少种方法？

假设方法总共有 $f(n)$，分成两种情况：①第一次放 1 格，剩下 $n-1$ 个格子，方法有 $f(n-1)$ 种；②第一次放 2 格，剩下 $n-2$ 个格子，方法有 $f(n-2)$ 种。这也是一个 Fibonacci 数列。

8.3.5　母函数

本节介绍一种求解递推关系的特殊思路——母函数。母函数（Generating Function，又译为生成函数）是算法竞赛中经常使用的一种解题方法，它用代数方法解决组合计数问题，是数学与应用的有趣结合。

本节尝试引导读者理解母函数，并用来解决一些算法问题。不过，读者仍然需要进一步深入地学习母函数的数学思想，这样才能更好地应用它。建议读者阅读一些组合数学方面的书籍，做一些习题，以加强理解[②]。

---

① 很多人说“黄金分割美学”其实是夸大其词。

② 《组合数学》，Richard A. Brualdi 著，冯舜玺译，机械工业出版社。

1. 整数划分

在讲解母函数之前先思考一个经典问题——整数划分。整数划分是指把一个正整数 n 分解成多个整数的和，这些数大于等于 1、小于等于 n。不同划分法的总数叫作划分数。例如 $n=4$ 时有 5 种划分，即{1,1,1,1}、{1,1,2}、{2,2}、{1,3}、{4}。

这个问题有很多扩展[①]，例如将 n 划分成最大数不超过 m 的划分数，$m \leqslant n$。当 $n=4$，$m=2$ 时有 3 种划分，即{1,1,1,1}、{1,1,2}、{2,2}。

> **hdu 1028 "Ignatius and the Princess III"**
>
> 求整数 n 有多少种划分，$1 \leqslant n \leqslant 120$。
>
> 输入一个数字 n，输出划分数。

在引入母函数方法之前先用递归求解，代码如下，其中函数 part(n,n)返回对 n 划分的结果。

递归求整数划分

```
#include <bits/stdc++.h>
using namespace std;
int part(int n, int m) {                              //将 n 划分成最大数不超过 m 的划分数
    if(n == 1||m == 1)    return (1);
    else if(n < m)       return part(n,n);
    else if(n == m)       return 1 + part(n,n-1);
    else        return part(n-m, m) + part(n, m-1);  //这一行导致 TLE
}
int main(){
    int n;
    while(cin >> n)
     cout << part(n, n) << endl;
    return 0;
}
```

函数 part()的最后一行有两种情况。

(1) part$(n-m,m)$：划分中有一个数为 m，那么从 n 中减去 m，继续对 $n-m$ 进行划分；

(2) part$(n,m-1)$：划分中每个数都小于 m，即每个数不大于 $m-1$，继续划分。

但是，用上面的递归代码提交 hdu 1028 题，结果是 TLE。观察程序的最后一行，发现递归翻倍，是 $O(2^n)$的复杂度。

用 DP 可以显著降低复杂度。把递归程序中的逻辑改写成递推式，在函数 part()中提前预计算出所有 n 的划分数。程序的计算复杂度是 $O(n^2)$。

用 DP 求整数划分

```
const int MAXN = 200;
int dp[MAXN + 1][MAXN + 1];            //dp[n][m]: 将 n 划分成最大数不超过 m 的划分数
```

① 扩展的划分问题：http://www.cnblogs.com/radiumlrb/p/5797168.html(永久网址：perma.cc/92XR-N9DF)。

```
void part() {                          //预计算 dp[n][m],求出所有 n 的划分
    for(int n = 1; n <= MAXN; n++)
        for(int m = 1; m <= MAXN; m++){
            if((n == 1)||(m == 1))     dp[n][m] = 1;
            else if(n < m)             dp[n][m] = dp[n][n];
            else if(n == m)            dp[n][m] = dp[n][m - 1] + 1;
            Else                       dp[n][m] = dp[n][m - 1] + dp[n - m][m];
        }
}
```

下面用母函数方法求解整数划分问题。

2. 母函数的概念

在解决整数划分问题之前先通过一个更简单的问题介绍母函数的概念。


视频讲解

问题：从数字 1、2、3、4 中取出一个或多个相加(每个数最多只能用一次)，能组合成几个数？每个数有几种组合？

在表 8.3 中，第 1 行是组合得到的数字，第 2 行是组合的情况，第 3 行是有几种组合。

表 8.3 数字组合问题

数字 S	1	2	3	4	5	6	7	8	9	10
组合	1	2	1+2 3	1+3 4	1+4 2+3	1+2+3 2+4	1+2+4 3+4	1+3+4	2+3+4	1+2+3+4
数量 N	1	1	2	2	2	2	2	1	1	1

下面引进一个公式，并把公式展开，这个公式能解决上面的数字组合问题。后文会介绍这个公式是怎么来的。

$$(1+x)(1+x^2)(1+x^3)(1+x^4)=1+x+x^2+2x^3+2x^4+2x^5+2x^6+2x^7+x^8+x^9+x^{10}$$

读者仔细观察，可以发现公式和上面的表是有关系的：

(1) 公式左边的 x 的幂与组合用到的数字 1、2、3、4 相对应。观察公式左边，包括 4 个部分，$(1+x)$中的 x 是 1 次幂，$(1+x^2)$中的 x^2 是 2 次幂，依此类推，刚好是数字 1、2、3、4。

(2) 公式右边 x 的幂与表格中的组合数 S 是对应的。公式右边 x 的幂从 1 到 10，组合数 S 也从 1 到 10。

(3) 公式右边的系数与表格中的数量 N 相对应，都是 1、1、2、2、2、2、2、1、1、1。

因此，用这个公式可以计算上面的组合数问题。

这就是母函数的原理：**“把组合问题的加法与幂级数的乘幂对应起来”**。

那么，这个公式是如何得到的？

为了更容易理解，把公式左边写成以下的形式：

$$(1+x)(1+x^2)(1+x^3)(1+x^4)$$
$$=(x^{0\times1}+x^{1\times1})(x^{0\times2}+x^{1\times2})(x^{0\times3}+x^{1\times3})(x^{0\times4}+x^{1\times4})$$

其含义以公式右边的$(x^{0\times1}+x^{1\times1})$为例，$x^{0\times1}$表示不用数字 1，$x^{1\times1}$表示用数字 1。

所以,这个公式实际上就是组合问题的反映:**用或者不用数字 1、用或者不用数字 2、用或者不用数字 3**,依此类推,公式就是这样构造出来的。公式构造出来后,把它展开后的结果就是组合问题的答案。

母函数的定义:对于序列 $a_0, a_1, a_2, \cdots$,构造函数 $G(x)=a_0+a_1x+a_2x^2\cdots$,称 $G(x)$ 为序列 $a_0, a_1, a_2, \cdots$ 的母函数。

简单地说,母函数是一种幂级数,其中每一项的系数反映了这个序列的信息。在本例中,a_kx^k 的 a_k 是数 k 的组合数量。

3. 用母函数解决整数划分问题

整数划分比上面的数字组合问题复杂一些,因为整数划分的数字是可以重复的。可以这样设计整数划分的母函数:

$$(x^{0\times 1}+x^{1\times 1}+x^{2\times 1}+\cdots)(x^{0\times 2}+x^{1\times 2}+x^{2\times 2}+\cdots)(x^{0\times 3}+x^{1\times 3}+x^{2\times 3}+\cdots)\cdots$$
$$=(1+x+x^2+\cdots)(1+x^2+x^4+\cdots)(1+x^3+x^6+\cdots)\cdots$$

其中,$(x^{0\times 1}+x^{1\times 1}+x^{2\times 1}+\cdots)$ 的含义是**不用数字 1、用一次 1、用两次 1**,依此类推。

母函数展开后,第 x^n 项的系数就是数字 n 的划分数。

那么,如何编程计算母函数展开后的系数?模拟手工计算过程就可以了。首先把前两部分 $(1+x+x^2+\cdots)$ 和 $(1+x^2+x^4+\cdots)$ 相乘并展开;展开的结果再与第 3 部分 $(1+x^3+x^6+\cdots)$ 相乘并展开;继续这个过程直到完成。

用母函数求整数划分(hdu 1028)

```
const int MAXN = 200;
int c1[MAXN + 1], c2[MAXN + 1];
void part() {
    int i, j, k;
    for(i = 0; i <= MAXN; i++){              //初始化,即第 1 部分(1 + x + x² + …)的系数,都是 1
        c1[i] = 1;   c2[i] = 0;
    }
    for(k = 2; k <= MAXN; k++){              //从第 2 部分(1 + x² + x⁴ + …)开始展开
        for(i = 0; i <= MAXN; i++)
        //k = 2 时,i 循环第 1 部分(1 + x + x² + …),j 循环第 2 部分(1 + x² + x⁴ + …)
            for(j = 0; j + i <= MAXN; j += k)
                c2[i + j] += c1[i];
        for(i = 0; i <= MAXN; i++) {         //更新本次展开的结果
            c1[i] = c2[i];  c2[i] = 0;
        }
    }
}
```

数组第 c1[n]项用来记录每次展开后第 x^n 项的系数,计算结束后,c1[n]就是整数 n 的划分数。数组 c2[]用于记录临时计算结果。

上面的代码可以当成模板,请读者仔细理解细节。虽然不同的问题有不同的母函数,但都是方程式的展开,代码和上面的差不多,只要做相应的修改即可。

本节讲解的是"普通型"母函数,可用于求组合方案数;还有一种"指数型"母函数,用于求排列数。例如{1,2,3,4},要求每个数字用且只用一次,那么组合方案只有 1 种,而排列有

4!=24 种。

求组合方案的题目,如果能用普通型母函数求解[①],一般也能用 DP 求解。但是,众所周知,DP 的难点在于递推关系,想不到就做不出来;而母函数的思路是很直观的,容易理解。比如整数划分问题,母函数的方法要比 DP 简单一些。

4. 指数型母函数

先看一个典型的例题——hdu 1521 题。

hdu 1521 "排列组合"

有 n 种物品,并且知道每种物品的数量,求从中选出 m 件物品的排列数。例如有两种物品 A、B,并且数量都是 1,从中选两件物品,则排列有"AB"和"BA"两种。

输入:每组输入数据有两行,第 1 行是两个数 n 和 m($1 \leqslant m, n \leqslant 10$),表示物品数;第 2 行有 n 个数,分别表示这 n 件物品的数量。

输出:对应每组数据输出排列数(任何运算不会超出 2^31 的范围)。

分析题目,假设有 3 种物品 A、B、C,数量分别是 2、3、1,即{A,A,B,B,B,C},从中选两件物品,则排列是{AA,AB,BA,AC,CA,BB,BC,CB},共 8 种。

针对这个例子,直接给出指数型母函数的解决方案。下面表达式的第 1 行是母函数公式,第 2 行展开,第 3 行整理:

$$\begin{aligned} G(x) &= \left(1+\frac{x}{1!}+\frac{x^2}{2!}\right)\left(1+\frac{x}{1!}+\frac{x^2}{2!}+\frac{x^3}{3!}\right)\left(1+\frac{x}{1!}\right) \\ &= 1+3x+4x^2+\frac{19}{6}x^3+\frac{19}{12}x^4+\frac{1}{2}x^5+\frac{1}{12}x^6 \\ &= 1+\frac{3x}{1!}+\frac{8x^2}{2!}+\frac{19}{3!}x^3+\frac{38}{4!}x^4+\frac{60}{5!}x^5+\frac{60}{6!}x^6 \end{aligned}$$

第 1 行的 3 个括号内分别代表两个 A、3 个 B、1 个 C。

答案就隐含在最后一行中。例如$\frac{19}{3!}x^3$,x^3 的幂 3 表示选 3 件物品,系数 19 表示有 19 种排列。这一行给出了所有的答案:物品 A、B、C,数量分别有 2、3、1 个,那么选一件物品的排列有 3 种、选两件有 8 种、选 3 件有 19 种、选 4 件有 38 种、选 5 件有 60 种、选 6 件有 60 种。

是不是很神奇?下面分析母函数公式。

把公式写成$\frac{x}{1!}$、$\frac{x^2}{2!}$、$\frac{x^3}{3!}$这样的形式,实际上是在处理排列。例如,第 1 行的第 1 部分$\left(1+\frac{x}{1!}+\frac{x^2}{2!}\right)$是对物品 A(有两个 A)的排列。为了容易理解,可以改写成$\left(\frac{1x^0}{0!}+\frac{1x^1}{1!}+\frac{1x^2}{2!}\right)$,意思如下:

$\frac{1x^0}{0!}$,不选 A 的排列有 1 种,即{ϕ};

$\frac{1x^1}{1!}$,选 1 件 A 的排列有 1 种,即{A};

① 这里对整数划分给出了有趣的解释:《组合数学》,Richard A. Brualdi 著,机械工业出版社,8.3 节,分拆数。

$\frac{1x^2}{2!}$，选两件 A 的排列有 1 种，即{AA}。

同理，选 B 物品(有 3 个 B)时计算公式是$\left(1+\frac{x}{1!}+\frac{x^2}{2!}+\frac{x^3}{3!}\right)$，选 C 物品(有 1 个 C)时计算公式是$\left(1+\frac{x}{1!}\right)$。

当同时选多个物品时，把公式相乘，其展开项就是排列情况。例如选 A、B 两种物品，A 的$\frac{x}{1!}$与 B 的$\frac{x^3}{3!}$相乘，表示选 1 个 A，再选 3 个 B 的排列数量，计算得到$\frac{x}{1!}\times\frac{x^3}{3!}=\frac{x^4}{6}=\frac{4x^4}{4!}$，分子系数是 4，表示有 4 种排列，它们是{ABBB，BABB，BBAB，BBBA}。

为什么要将分母写成 1!、2!、3! 这样的形式？它体现了排列和组合的关系：k 个物品的排列和 k 个物品的组合相差 k! 倍。在选多个物品时，利用这个特点可以处理多重组合的排列问题。

例如 A 有两个、B 有 3 个，组合只有一种，是{A，A，B，B，B}，下面求排列数。

(1) 两个 A 的排列公式是$\frac{x^2}{2!}$，分母的 2! 处理了排列的问题：如果是两个不同的 A_1、A_2，应该有两种排列，即{A_1A_2，A_2A_1}，但是 A_1、A_2 相同，所以需要除以 2!，得到一种排列{AA}。

(2) 3 个 B 的排列公式是$\frac{x^3}{3!}$，分析是一样的，分母除以 3!，剔除重复的排列，得到一种排列{BBB}。

(3) 合起来排列公式是$\frac{x^2}{2!}\times\frac{x^3}{3!}=\frac{x^5}{12}=\frac{10x^5}{5!}$，分子的系数是 10，表示有 10 种排列{AABBB，ABABB，ABBAB，…}。

现在给出指数型母函数的定义：对序列 $a_0,a_1,a_2,\cdots$，构造函数 $G(x)=a_0+\frac{a_1}{1!}x+\frac{a_2}{2!}x^2+\frac{a_3}{3!}x^3\cdots$，称 $G(x)$ 为序列 $a_0,a_1,a_2,\cdots$ 的指数型母函数。

指数型母函数的程序和普通型母函数的程序非常相似，只多了对分母 k! 的处理。

hdu 1521 题的程序留给读者自己编写。

8.3.6 特殊计数

1. Catalan 数

1) 定义

Catalan 数是一个数列，它的一种定义如下：

$$C_n=\frac{1}{n+1}\binom{2n}{n},\quad n=0,1,2,\cdots$$

前一部分 Catalan 数是 1，1，2，5，14，42，132，429，1430，4862，16796，58786，208012，742900，2674440，9694845，35357670…Catalan 数的增长速度极快。

Catalan 数看起来有点奇怪，但是观察它的公式，其中有组合计数。实际上，Catalan 数

是很多组合计数应用问题的数学模型，是一个很常见的数列[①]。

Catalan 数有以下两种基本模型。

模型Ⅰ：$C_n=\frac{1}{n+1}\binom{2n}{n}=\binom{2n}{n}-\binom{2n}{n+1}=\binom{2n}{n}-\binom{2n}{n-1}$

其中，$\binom{2n}{n}$是在 $2n$ 种情况中选 n 个的组合数；$\binom{2n}{n-1}$是在 $2n$ 种情况中选 $n-1$ 个的组合数。注意，$\binom{2n}{n-1}$和$\binom{2n}{n+1}$等价。

模型Ⅰ的公式可以从一个**基本模型**推导出来：把 n 个 1 和 n 个 0 排成一行，使这一行的任意前 k 个数中 1 的数量总是大于或等于 0 的数量(或者 0 的数量大于等于 1 的数量，二者等价)。这样的排列有多少个？答案是这样的排列一共有 C_n 个，即 Catalan 数。

模型Ⅱ：第Ⅱ种模型是递推。

$$C_n = C_0C_{n-1} + C_1C_{n-2} + \cdots + C_{n-2}C_1 + C_{n-1}C_0 = \sum C_kC_{n-k}, \quad C_0 = 1$$

下面几个应用场景可以按上面两个模型进行解释。

2）棋盘问题

hdu 2067 题。

hdu 2067“小兔的棋盘”

小兔的叔叔从外面旅游回来给它带来了一个礼物，小兔高兴地跑回自己的房间，拆开一看是一个棋盘，小兔有所失望。不过没过几天它发现了棋盘的好玩之处，从起点(0,0)走到终点(n,n)的最短路径数是$C(2n,n)$，现在小兔想如果不穿过对角线(但可接触对角线上的格点)，这样的路径数有多少？

题目的意思是一个 n 行 n 列的棋盘，从左下角走到右上角，一直在对角线右下方走，不穿过主对角线，走法有多少种？例如 $n=4$ 时有 14 种走法。

这个问题就是上面的基本模型(Ⅰ)，下面进行分析。

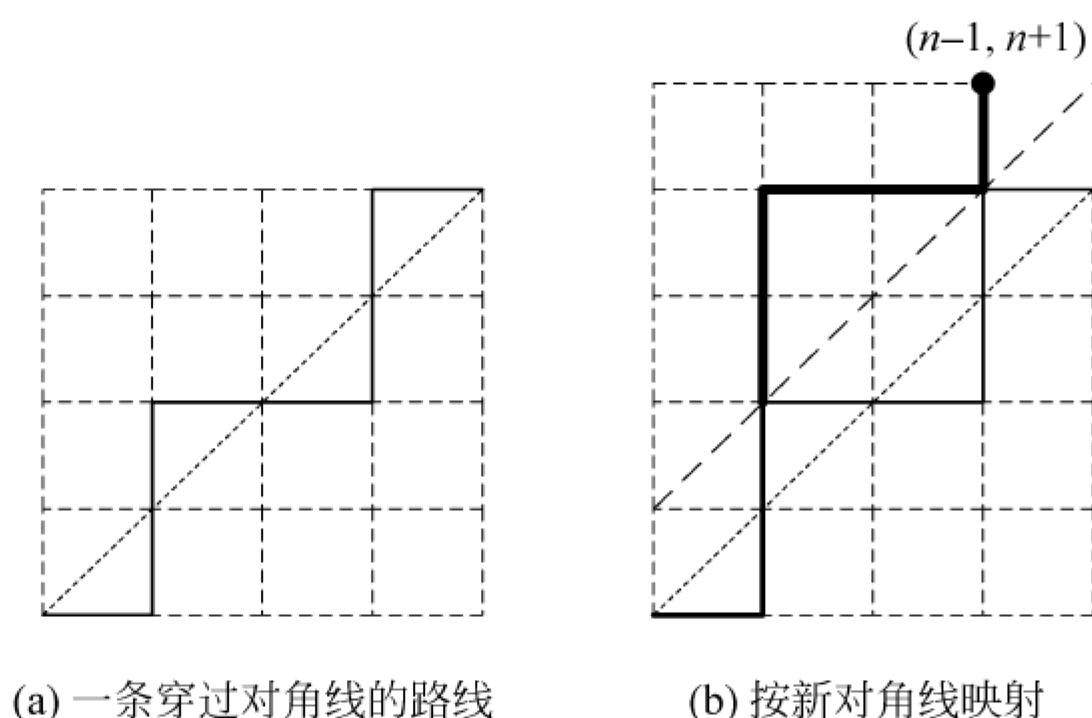
对方向编号，向上是 0，向右是 1，那么从左下角走到右上角一定会经过 n 个 1 和 n 个 0。满足要求的路线是走到任意一步 k，前 k 步中向右的步数(1 的个数)大于或等于向上的步数(0 的个数)，否则就穿过对角线了。

设从左下角走到右上角的总路线有 X 条，分成 3 个部分：对角线下面的 A 条路线，对角线上面的 B 条路线，穿过对角线的 C 条路线。不过，这 3 个部分可以简化为两个部分，即对角线下面的 A、穿过对角线的 Y(包括 B 和 C)。$A=X-Y$ 就是答案。

总路线 $X=\binom{2n}{n}$，它的意思是在 $2n$ 个位置放 n 个 1(剩下的 n 个肯定是 0)，这样的数有$\binom{2n}{n}$个。

① 这里列出了很多 Catalan 数的应用，注意看其中的棋盘问题：https://en.wikipedia.org/wiki/Catalan_number (短网址：t.cn/R1TgbAG)。

对于 Y,需要用到一种叫作 André's reflection method 的方法。图 8.1(a)给出了一条穿过对角线的路线(即 C 路线;或者给出一条在斜对角上方并不穿过对角线的路线,即 B 路线,分析和 C 路线一样)。在图 8.1(b)中,画一条新的对角线,把它画在原来对角线的上面一格。



(a) 一条穿过对角线的路线　　(b) 按新对角线映射

图 8.1　André's reflection method

下面开始操作:原来的路线,从左下角出发,第一次接触到这条新对角线后,把剩下的部分以新对角线为轴进行映射,得到新的路线。这条新的路线即图 8.1 中加粗的黑线。加粗黑线下面的一部分黑线是原来的,保持不变;上面一部分是新的,与原来那一部分对称。整个路线仍然是连续的,但是路线的终点变为$(n-1,n+1)$。注意,"在原对角线右下方不穿过主对角线的走法",即前文提到的 A 部分,与新对角线无交集,无法映射,被排除在外。

新的路线和原来的路线是一一对应的。这些新路线有多少个?此时有 $n+1$ 个 0、$n-1$ 个 1,共 $2n$ 个;选出 $n-1$ 个 1(等价于选出 $n+1$ 个 0)的排列有 $Y=\binom{2n}{n-1}$个。

因此 $A=X-Y=\binom{2n}{n}-\binom{2n}{n-1}$。

3) 括号问题

括号问题;用 n 个左括号和 n 个右括号组成一串字符串有多少种合法的组合?例如,"( ) ( ) (( ))"是合法的,而"( )) (( )"是非法的。显然,合法的括号组合是:任意前 k 个括号组合,左括号的数量大于等于右括号的数量。

定义左括号为 0、右括号为 1。问题转化为 n 个 0 和 n 个 1 组成的序列,在任意前 k 个序列中 0 的数量都大于等于 1 的数量。模型和上面的棋盘问题一样。

读者可以练习 hdu 5184 题:给定初始的括号序列,再给定 n 表示序列的总长度,问一共有多少种括号组成方式?

4) 出栈序列问题

给定一个以字符串形式表示的入栈序列,求出一共有多少种可能的出栈顺序?比如入栈序列为{1 2 3},则出栈序列一共有 5 种,即{1 2 3}、{1 3 2}、{2 1 3}、{2 3 1}、{3 2 1}。

分析可知,合法的序列是对于出栈序列中的每一个数字,在它后面的比它小的所有数字一定是按递减顺序排列的。例如,{3 2 1}是合法的,3 出栈之后,比它小的后面的数字是{2 1},且这个顺序是递减顺序;而{3 1 2}是不合法的,因为在 3 后面的数字{1 2}是一个递

增的顺序。

对于每一个数来说，必须进栈一次、出栈一次。定义进栈操作为0、出栈操作为1。n个数的所有状态对应n个0和n个1组成的序列。出栈序列，即要求进栈的操作数大于等于出栈的操作数。问题转化为由n个1和n个0组成的$2n$位二进制数，任意前k个序列中0的数量大于或等于1的数量。结果仍然是Catalan数。

hdu 1023题：火车进站、出站，模拟进栈和出栈操作。由于要计算第100个Catalan数，这个数非常大，需要用大数计算，读者可以用Java编程。

5）二叉树问题

n个结点构成的二叉树共有多少种情况？

例如有3个结点（图中的黑点）的二叉树，可以构成5种二叉树，如图8.2所示。

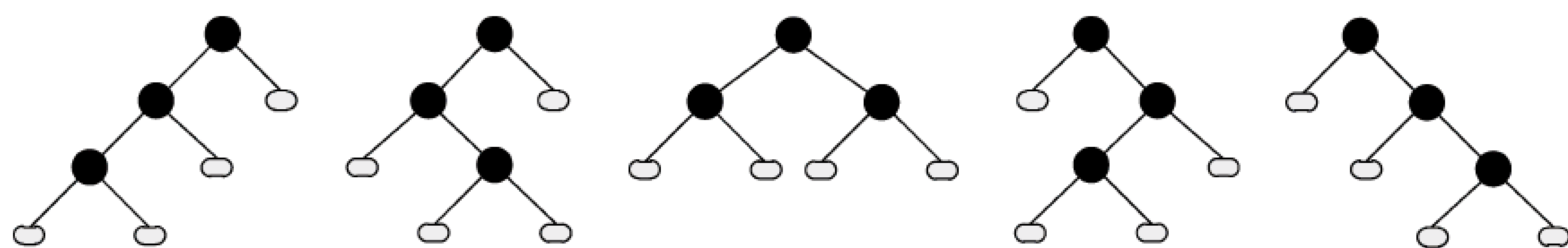


图8.2　包括3个结点的二叉树

这个问题符合模型Ⅱ：

$$C_n = C_0C_{n-1} + C_1C_{n-2} + \cdots + C_{n-2}C_1 + C_{n-1}C_0 = \sum C_kC_{n-k},\quad C_0 = 1$$

其含义如下：

C_0C_{n-1}：右子树有0个结点＋左子树有$n-1$个结点；

C_1C_{n-2}：右子树有1个结点＋左子树有$n-2$个结点；

⋮

$C_{n-1}C_0$：右子树有$n-1$个结点＋左子树有0个结点。

读者可以练习hdu 1130/3240题。

6）其他问题

买票找零问题。

三角剖分问题：把一个凸多边形内部划分成多个三角形有多少种方法？

7）编程计算Catalan数

有多种计算方法：

(1) $C_n = C_0C_{n-1} + C_1C_{n-2} + \cdots + C_{n-2}C_1 + C_{n-1}C_0 = \sum C_kC_{n-k},\quad C_0 = 1$

(2) $C_n = \dfrac{4n-2}{n+1}C_{n-1},\quad C_0 = 1$

(3) $C_n = \dfrac{1}{n+1}\dbinom{2n}{n} = \dfrac{(2n)!}{(n+1)!n!}$

从公式(2)可知，当n很大时，$C_n/C_{n-1}\approx 4$。所以Catalan数是以约4^n递增的，增长极快。

这3个公式的应用场合不同。

用公式(1)的场合：需要输出Catalan数的值。此时n较小，例如算$n\leqslant 100$内的Catalan数，不过Catalan数仍然是一个超级大的数。此时用公式(1)比用公式(2)好。因为公式(2)需要算大数的乘/除法，它比公式(1)的递推公式更容易溢出。例如hdu 2067题“小

兔的棋盘”,读者可以分别用两种方法编程。可以发现,如果只是简单地用 int64 来定义 Catalan 数,当计算到第 34 个 Catalan 数时公式(2)计算出错,而公式(1)仍然正确。对于更大的 Catalan 数,需要进行高精度计算,例如 hdu 1023/1130 题,计算第 100 个 Catalan 数。

用公式(2)、(3)的场合:n 非常大,不能直接输出 Catalan 数,而是做取模操作。例如 hdu 5184,需要算第 10 万个 Catalan 数,用公式(1)算太慢了。此时用公式(2)是很好的选择。不过,(2)和(3)都有大数除法,对大数做除法会损失精度,所以需要转换为逆元,然后再取模。如果用公式(3)算,注意先预计算 n 的阶乘(算阶乘的同时对阶乘取模),然后再用公式计算。

【习题】

除了上面的基础题外,读者可练习下面的题目:

hdu 4828,卡特兰数,逆元。

hdu 5673,卡特兰数,逆元。

hdu 5177,$n \leqslant 10^{18}$ 的卡特兰数。

2. Stirling 数

Stirling 数也是解决特定组合问题的数学工具,包括两种,即第一类 Stirling 数和第二类 Stirling 数,它们有相似的地方。

首先通过一个经典的仓库钥匙问题来了解第一类 Stirling 数。

问题描述:有 n 个仓库,每个仓库有两把钥匙,共 $2n$ 把钥匙,有 n 位保管员。

问题 1:如何放钥匙使得保管员都能够打开所有仓库?

问题 2:保管员分别属于 k 个不同的部,部中的保管员数量和他们管理的仓库数量一样多,例如第 i 个部有 m 个管理员,管 m 个仓库。如何放钥匙,使得同部的所有保管员能打开本部的所有仓库,但是无法打开其他的仓库?

问题 1 很好解答。1 号仓库放 2 号仓库的钥匙,2 号仓库放 3 号仓库的钥匙,依此类推,n 号仓库放 1 号仓库的钥匙,相当于 n 个仓库形成了一个闭环的圆;然后每个保管员拿一把钥匙即可,他打开一个仓库后就能拿到下一把钥匙,继续打开其他所有的仓库。

问题 2 是问题 1 的扩展:把 n 个仓库分成 k 个圆排列,每个圆内部按问题 1 处理。这里的麻烦问题是:把 n 个仓库分配到 k 个圆里,不能有空的圆,共有多少种分法?答案就是第一类 Stirling 数。

1) 第一类 Stirling 数

定义第一类 Stirling 数 $s(n,k)$:把 n 个不同的元素分配到 k 个圆排列里,圆不能为空。问有多少种分法?

下面直接给出第一类 Stirling 数的递推公式[①]:

$$s(n,k) = s(n-1,k-1) + (n-1)s(n-1,k), \quad 1 \leqslant k \leqslant n$$
$$s(0,0) = 1, \quad s(k,0) = 0, \quad 1 \leqslant k \leqslant n$$

① 《组合数学》,Richard A. Brualdi 著,机械工业出版社。第 8 章,定理 8.2.9,推导了第一类 Stirling 数的递推公式。

根据递推公式计算部分 Stirling 数，如表 8.4 所示。

表 8.4 第一类 Stirling 数 $s(n,k)$ 的值

n \ k	0	1	2	3	4	5	6	…
0	1							
1	0	1						
2	0	1	1					
3	0	2	3	1				
4	0	6	11	6	1			
5	0	24	50	35	10	1		
6	0	120	274	225	85	15	1	
…								

例如：

$s(2,1)=1$，两个物体 a、b 放在 1 个圆圈里，有 1 种方案，即{(ab)}；

$s(3,1)=2$，3 个物体 a、b、c 放在 1 个圆圈里，有两种方案，即{(abc)}和{(acb)}；

$s(3,2)=3$，3 个物体 a、b、c 放在两个圆圈里，有 3 种方案，即{(ab),(c)}、{(ac),b}、{(a),(bc)}。

2) 第二类 Stirling 数

定义第二类 Stirling 数 $S(n,k)$：把 n 个不同的球分配到 k 个相同的盒子里[①]，不能有空盒子。问有多少种分法？

$S(n,k)$ 的递推公式如下：

$$S(n,k) = kS(n-1,k) + S(n-1,k-1), \quad 1 \leqslant k \leqslant n$$

$$S(0,0) = 1, \quad S(i,0) = 0, \quad 1 \leqslant i \leqslant n$$

根据递推公式计算部分 Stirling 数，如表 8.5 所示。

表 8.5 第二类 Stirling 数 $S(n,k)$ 的值

n \ k	0	1	2	3	4	5	6	…
0	1							
1	0	1						
2	0	1	1					
3	0	1	3	1				
4	0	1	7	6	1			
5	0	1	15	25	10	1		
6	0	1	31	90	65	15	1	
…								

① 读者自然能想到，根据球是否一样、盒子是否相同、盒子是否可为空可以组合成各种类似的问题，例如把 n 个一样的球分配到 k 个相同的盒子里、把 n 个一样的球分配到 k 个不同的盒子里，等等。在这些问题中，第二类 Stirling 数比较复杂，但它是很基本的问题。所有的情况参考《应用组合数学》，Fred S. Roberts、Barry Tesman 著，冯速译，机械工业出版社，2.10 节，分装问题；公式的推导见 5.5.3 节。

例如：

$S(2,1)=1$，两个球 a、b 放在 1 个盒子里，有 1 种方案，即{(ab)}；

$S(3,1)=1$，3 个球 a、b、c 放在 1 个盒子里，有 1 种方案，即{(abc)}；

$S(3,2)=3$，3 个球 a、b、c 放在两个相同的盒子里，有 3 种方案，即{(ab),(c)}、{(ac),b}、{(a),(bc)}。

【习题】

hdu 4372 "Count the Buildings"，第一类 Stirling 数。

hdu 2643 "Rank"，第二类 Stirling 数。

8.4 概率和数学期望

概率和数学期望是概率论和统计学中的数学概念。设有随机变量 X，出现取值 x_i 的概率是 p_i，把它们的乘积之和称为数学期望(Expected Value，或者均值 mean)，记为 $E(X)$：

$$E(X)=\sum_{i=1}^{n} x_i p_i$$

$E(X)$是基本的数学特征之一，它反映了随机变量平均值的大小。

以妇女的生育率为例，假设某国有 2000 万个育龄妇女，不生育妇女有 277 万，一孩 724 万，二孩 883 万，三孩 116 万。记一个妇女的孩子数量是 X，取值 0、1、2、3，概率分别是 277/2000=0.1385、724/2000=0.362、883/2000=0.4415、116/2000=0.058。那么平均每个妇女生育的孩子数量如下：

$$E(X)=0\times 0.1385+1\times 0.362+2\times 0.4415+3\times 0.058=1.419$$

数学期望具有线性性质。有限个随机变量之和的数学期望等于每个变量的数学期望之和：

$$E(X+Y)=E(X)+E(Y)$$

竞赛中求数学期望的题目一般都会用到它的线性性质。由于线性性质和 DP 的状态转移思想很相似，所以常常用 DP 来实现。

1. 例题 1

首先看一个简单的例题。

poj 2096 "Collecting Bugs"

一个软件有 s 个子系统，会产生 n 种 bug。现在要找出所有种类的 bug。假设某人一天发现一个 bug。一个 bug 属于某个子系统的概率是 $1/s$，属于某种分类的概率是 $1/n$。问发现 n 种 bug，且每个子系统都发现 bug 的天数的期望。$0<n,s\leqslant 1000$。

输入：n 和 s；

输出：数学期望。

输入样例：
1 2
输出样例：
3.0000

定义状态 dp[i][j]，它表示已经找到 i 种 bug，并存在于 j 个子系统中，要达到目标状态还需要的期望天数。其中，dp[n][s]表示已经找到 n 种 bug，且存在于 s 个子系统，说明已经达到了目标，还需要 0 天，所以 dp[n][s]=0。从 dp[n][s]倒推回 dp[0][0]，就是本题的答案，即还没有找到任何 bug 的情况下到达 dp[n][s]时需要的期望天数。

从 dp[i][j]开始：后面 1 天找到 1 个 bug，可能有以下 4 种情况。

(1) dp[i][j]：发现一个 bug，属于已经有的 i 个分类和 j 个系统，概率为 p1=(i/n)*(j/s)。这一天相当于浪费了。

(2) dp[i+1][j]：发现一个 bug，不属于已有分类、属于已有系统，概率为 p2=(1−i/n)*(j/s)。

(3) dp[i][j+1]：发现一个 bug，属于已有分类、不属于已有系统，概率为 p3=(i/n)*(1−j/s)。

(4) dp[i+1][j+1]：发现一个 bug，不属于已有系统、不属于已有分类，概率 p4=(1−i/n)*(1−j/s)。

可以验证：p1+p2+p3+p4=1。

状态转移方程如下：

$$\begin{aligned} dp[i][j] = & p1 * dp[i][j] + p2 * dp[i+1][j] + p3 * dp[i][j+1] + \\ & p4 * dp[i+1][j+1] + 1 \quad // \text{末尾加上 1 天} \end{aligned}$$

整理得到：

$$\begin{aligned} dp[i][j] &= (p2 * dp[i+1][j] + p3 * dp[i][j+1] + p4 * dp[i+1][j+1] + 1)/(1-p1) \\ &= (n * s + (n-i) * j * dp[i+1][j] + i * (s-j) * dp[i][j+1] + \\ &\quad (n-i) * (s-j) * [i+1][j+1])/(n * s - i * j) \end{aligned}$$

在写程序时，从 dp[n][s]倒推到 dp[0][0]，dp[0][0]就是答案。

poj 2096 部分程序

```
cin >> n >> s;
for (int i = n; i >= 0; i-- )
  for (int j = s; j >= 0; j-- ){
    if ( i == n && j == s )
      dp[n][s] = 0.0;
    else
      dp[i][j] = (n * s + (n - i) * j * dp[i + 1][j] + i * (s - j) * dp[i][j + 1]
                + (n - i) * (s - j) * dp[i + 1][j + 1]) /(n * s - i * j);
  }
```

2. 例题 2

hdu 4035 是经典的迷宫概率问题，综合了图、数学期望、DP 等内容。

hdu 4035 "Maze"

一个迷宫有 n 个房间，用 $n-1$ 条隧道连通起来。每个房间里都有陷阱和逃生口。某人的起点在房间 1，在每个房间都有 3 种可能：

(1) 落入陷阱被杀死，回到房间 1，概率为 k_i；

(2) 找到逃生口，走出迷宫，概率为 e_i；

(3) 在该房间连接的隧道中随机走一条，进入下一个房间。

求逃出迷宫所要走的隧道数量的期望值。

首先分析这个迷宫，它是一棵树。

一个有 n 个点、$n-1$ 条边的无向连通图，图上肯定没有回路，这样的图是一棵树。证明如下：用反证法，假设有一个回路，那么在这个回路上可以删除一条边而不影响整体的连通；删除之后，还有 n 个点、$n-2$ 条边，这是不可能连通的。

要使有 n 个点的图是连通的，至少需要 $n-1$ 条边。生成一个连通图，可以用扩大路径法，从一个点开始，每加入一个新的点，至少需要一条边来连接，所以 n 个点至少需要 $n-1$ 条边才能连通。

下面是推导过程和编程思路。

1) 定义 DP 状态 $E[i]$

在结点 i 处，逃出迷宫所要走的边数的期望。

$E[1]$就是所求的答案。

根据树的特点，分析 $E[i]$：

i 是叶子结点，即 i 没有子结点。在结点 i 有 3 种情况，即被杀、逃出、回到父结点。

$$E[i] = k_i * E[1] + e_i * 0 + (1 - k_i - e_i) * (E[\text{father}[i]] + 1) \tag{8-1}$$

i 是非叶子结点，设 i 连接的边数是 m，有 3 种情况，即被杀、逃出、转到其他结点。

$$E[i] = k_i * E[1] + e_i * 0 + (1 - k_i - e_i)/m * (E[\text{father}[i]] + 1 + \sum(E[\text{child}[i]] + 1)) \tag{8-2}$$

2) 计算过程

设对于每个结点：

$$E[i] = A_i * E[1] + B_i * E[\text{father}[i]] + C_i$$

目标是求 $E[1]$，$E[1]=A_1 * E[1]+B_1 * 0+C_1$，即 $E[1]=C_1/(1-A_1)$。

在叶子结点上有：

$$A_i = k_i$$
$$B_i = 1 - k_i - e_i$$
$$C_i = 1 - k_i - e_i$$

在非叶子结点上，设 j 为 i 的子结点，则：

$$\begin{aligned}\sum(E[\text{child}[i]]) &= \sum E[j] \\ &= \sum(A_j * E[1] + B_j * E[\text{father}[j]] + C_j) \\ &= \sum(A_j * E[1] + B_j * E[i] + C_j)\end{aligned}$$

代入式(8-1)和式(8-2)中,消去 $E[child[i]]$ 和 $E[father[j]]$,可以得到非叶子结点的 A_i、B_i、C_i 的表达式。

3) 编程思路

从上面的推导过程可知,计算过程是从叶子结点开始算,再算它们的父结点,直到算出根结点的 A_1、B_1、C_1,得到 $E[1]$。在编程时,需要按从叶子结点到根结点的顺序遍历每个点。

这个过程用 DFS 编程是最合适的。从根结点 1 出发,用 DFS 遍历整棵树;DFS 到最底层的叶子结点时,计算叶子结点的 A_i、B_i、C_i,然后逐步回退,再计算非叶子结点的 A_i、B_i、C_i。

在题目中,图的规模 $n \leqslant 10\,000$,需要用邻接表存储。请读者在学习第 10 章的相关内容后再回头做这一题。

【习题】

hdu 3853 "LOOPS",基础题。

hdu 4405 "Aeroplane chess",简单题。

poj 3071 "Football",简单概率 DP。

poj 3744 "Scout YYF I",用矩阵优化求概率。

hdu 4089 "Activation",2011 年北京区域赛题目,概率 DP,较难。

8.5　公平组合游戏

本节讨论的公平组合游戏(Impartial Combinatorial Game,ICG)[①]是满足以下特征的一类问题:



视频讲解

(1) 有两个玩家,游戏规则对两人是公平的;

(2) 游戏的状态有限,能走的步数也有限;

(3) 两人轮流走步,当一个玩家不能走步时游戏结束;

(4) 游戏的局势不能区分玩家身份,像围棋这样有黑、白两方的游戏就不属于此类问题。

ICG 问题有一个特征:给定初始局势,并且指定先手玩家,如果双方都采取最优策略,那么获胜者就已经确定了。也就是说,ICG 问题存在必胜策略。

本节讲解 ICG 问题的必胜策略,有关的知识点有 P-position、N-position、Nim Game、Sprague-Grundy 函数、威佐夫游戏等。ICG 很早就得到了研究,例如对于 Nim Game 问题,1902 年 C. Bouton 在一本著作中进行了分析;对于 Sprague-Grundy 函数,由数学家 Grundy 和 Sprague 在 1930 年分别独立发现。

---

① 在算法竞赛中,常常称这类问题是"博弈论"问题。虽然 Nim Game、Sprague-Grundy 函数也属于博弈论的范畴,不过在普通的博弈论教材中并不能找到有关内容。在一些应用组合数学书中会提到有关知识,请参考《应用组合数学》,Alan Tucker 著,冯速译,人民邮电出版社,第 11 章。

Sprague-Grundy 函数是本节最重要的内容。

8.5.1 巴什游戏与 P-position、N-position

首先给出一个简单的例题。小学奥数中有这样的题目。

1. 巴什游戏(Bash Game)

> **hdu 1846 "Brave Game"**
>
> 有 n 颗石子,甲先取,乙后取,每次可以拿 1～m 颗石子,轮流拿下去,拿到最后一颗的人获胜。
>
> 输入:n 和 m,$1\leqslant n,m\leqslant 1000$。
>
> 输出:如果先拿的甲赢了,输出"first",否则输出"second"。

程序非常简单,若 $n\%(m+1)==0$,则先手败,否则先手胜。

```
cin >> n >> m;
if(n % (m+1) == 0)    printf("second\n");
else                  printf("first\n");
```

分析如下:

(1) 当 $n\leqslant m$ 时,由于一次最少拿 1 个、最多拿 m 个,甲可以一次拿完,先手赢。

(2) 当 $n=m+1$ 时,无论甲拿走多少个(1～m 个),剩下的都多于 1 个、少于等于 m 个,乙都能一次拿走剩余的石子,后手取胜。

上面两种情况可以扩展为以下两种情况:

(Ⅰ) 如果 $n\%(m+1)=0$,即 n 是 $m+1$ 的整数倍,那么不管甲拿多少,例如 k 个,乙都拿 $m+1-k$ 个,使得剩下的永远是 $m+1$ 的整数倍,直到最后的 $m+1$ 个,所以后拿的乙一定赢。

(Ⅱ) 如果 $n\%(m+1)!=0$,即 n 不是 $m+1$ 的整数倍,还有余数 r,那么甲拿走 r 个,剩下的是 $m+1$ 的倍数,这样就转移到了情况(Ⅰ),相当于甲、乙互换,结果是甲赢。

在这个拿石子的游戏里,对于后拿的乙来说是很不利的,只有在 $n\%(m+1)=0$ 的情况下乙才能赢,在其他情况下都是甲赢。

2. P-position、N-position 与动态规划

上面对巴什游戏的解答虽然很好理解,但是如果稍作扩展,就不那么容易了。例如取石子的数量,不是 1～m 内的连续数字,而是只能在$\{a_1,a_2,\cdots,a_k\}$中选。对于此类问题,有必要研究一种通用的方法。

定义 P-position 为前一个玩家(Previous Player,即刚走过一步的玩家)的必胜位置、N-position 为下一个玩家(Next Player)的必胜位置。

当前状态是 N-position,表示马上走下一步的先手必胜;P-position 表示先手必败。

设只能拿数量为$\{1,4\}$的石头。在表 8.6 中,x 是石头的数量,pos 是对应的 position。

表 8.6 只能拿数量为{1,4}的石头

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	…
pos	P	N	P	N	N	P	N	P	N	N	P	N	P	N	…

表中的 pos 是这样计算的：

(1) $x=0,1,2,3,4$ 时，pos＝P,N,P,N,N。特别注意 $x=0$，即没有石头的情况，可以看成下一个玩家(先手玩家)没有石头可拿，输了，pos＝P。$x=1$ 时，先手玩家必赢，pos＝N。$x=2$ 时，先手只能拿 1 个，后手拿剩下的 1 个，后手赢，pos＝P。

(2) $x=5$ 时分两种情况：如果先手玩家拿 1 个，退回到 $x=5-1=4$ 的情况，此时后手玩家处于 N，即后手处于赢的位置；如果先手拿 4 个，退回到 $x=5-4=1$ 的情况，此时后手仍然处于 N。在两种情况下后手都赢了。所以 $x=5$ 时，pos＝P，即先手必输。

(3) $x=6$ 时，分别退回到 $x=6-1=5$ 和 $x=6-4=2$ 的情况，后手都处于 P。在两种情况下，后手都输了。所以 $x=6$ 时 pos＝N，先手必赢。

(4) $x=7$ 时略。

(5) $x=8$ 时：退回到 $x=8-1=7$，后手处于 P；退回到 $x=8-4=4$，后手处于 N。在后手有输有赢的情况下，先手肯定选让对方必败的方案，所以 $x=8$ 时 pos＝N。

可以观察到 pos 值是周期性变化的，周期为 5。

下面再举一个例子，设只能拿数量为{1,3,4}的石头，请读者验证表 8.7。

表 8.7 只能拿数量为{1,3,4}的石头

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
pos	P	N	P	N	N	N	N	P	N	P	N	N	N	N	P

pos 仍然是周期变化的，周期是 7。

上面的计算过程符合**动态规划**的思路。在编程时可以用动态规划，也可以直接按周期性变化规律做求余计算，hdu 1846 是一种最简单的情况，用求余编程计算就可以了。

巴什游戏有一些变形。例如 hdu 2147 “kiki’s game”，给出一个 $n\times m$ 的矩阵，从右上角走到左下角，看谁先到终点。画出 P-N 图，找到规律即可。

8.5.2 尼姆游戏

巴什游戏只有一堆石头，如果扩展到多堆石头，情况将复杂得多，这就是尼姆游戏(Nim Game)[①]。

尼姆游戏的规则：有 n 堆石子，数量分别是$\{a_1,a_2,a_3,\cdots,a_n\}$，两个玩家轮流拿石子，每次从任意一堆中拿走任意数量的石子，拿到最后一个石子的玩家获胜。

以 3 堆石头为例，简单情况的胜负如下。

{0,0,0}、{0,1,1}、$\{0,k,k\}$：先手必败。

{1,1,1}、{1,1,2}、{1,1,3}：先手必胜。

① https://en.wikipedia.org/wiki/Nim。

对于任意的$\{a_1, a_2, a_3, \cdots, a_n\}$，尼姆游戏有一个极为简单的判断胜负的方法，即做异或运算。

定理8.1：

若$a_1 \oplus a_2 \oplus a_3 \oplus \cdots \oplus a_n \neq 0$，则先手必胜，记此时的状态为N-position；

若$a_1 \oplus a_2 \oplus a_3 \oplus \cdots \oplus a_n = 0$，则先手必败，记此时的状态为P-position。

例如3堆石头的数量分别是{5,7,9}，转化为二进制数后做异或运算，结果如下：

$$\begin{array}{c} 0101 \\ 0111 \\ 1001 \\ \hline 1011 \end{array}$$

异或运算的结果不等于0，先手必胜。

在数学中，二进制的异或运算也可以看成是统计每一位上1的总个数的奇偶性：如果这一位上有偶数个1，那么这一位的计算结果为0；如果有奇数个，计算结果为1。所以，尼姆游戏中的异或运算也被称为**Nim-sum运算**。

下面对定理8.1做简单的证明。

（1）必定能够从N-position转化到P-position。也就是说，先手处于必胜点N-position时可以拿走一些石子，让后手必败。读者可以先自己思考如何转化。下面是具体方法：任选一堆，例如第i堆，石头数量是k；对剩下的$n-1$堆做异或运算，设结果为H；如果H比k小，就把第i堆石头减少到H；这样操作之后，因为$H \oplus H = 0$，所以n堆石头的异或等于0。可以证明，总会存在这样的第i堆石头，而且可能有多种转化方案。下面例题hdu 1850的程序中的“if((sum ^a[i])<=a[i])”统计了所有方案。

（2）进入P-position后，轮到的下一个玩家，不管拿多少石子都会转移到N-position。因为任何一堆的数量变化，都会使得这一堆的二进制数至少有一位发生变化，导致异或运算的结果不等于0。也就是说，这一个玩家不管怎么拿石子都必败。

（3）在游戏过程中，按上述（1）和（2）的步骤在N-position和P-position之间交替转化，直到所有堆的石头都是0，即终止于P-position。

上述证明过程也说明了玩家该如何进行游戏。

hdu 1850 “Being a Good Boy in Spring Festival”

两人小游戏：桌子上有n堆扑克牌；每堆牌的数量分别为a_i；两人轮流进行；每走一步可以从任意一堆中取走任意张牌；桌子上的扑克牌全部取光，则游戏结束；最后一次取牌的人为胜者。问先手的人如果想赢，第一步有几种选择？

输入：n表示扑克牌的堆数；$a_i(i=1\sim n)$表示每堆扑克牌的数量。

输出：如果先手能赢，输出他第一步可行的方案数，否则输出0。

主要代码如下：

```
int sum = 0, ans = 0;                          //sum 是 Nim - sum,ans 是第一步可行的方案数
for(int i = 0; i<n; i++)    sum ^ = a[i];      //异或计算,求 Nim - sum
```

```
if(sum == 0)    cout << 0 << endl;                //开始局面是 P - position,先手必败
else{                                             //开始局面是 N - position,先手胜
  for(int i = 0; i < n; i++)
    if((sum ^a[i]) <= a[i])                       //计算第一步所有的可能方案
      ans++;
  cout << ans << endl;
}
```

程序中的"if((sum ^a[i]) <= a[i])"计算第一步的方案数,它利用了异或运算的原理:$A \oplus B \oplus B = A$。设 H 等于除了 $a[i]$之外其他所有数的异或,有:

$$\text{sum} = H \,\hat{}\, a[i]$$

$$\text{sum} \,\hat{}\, a[i] = H \,\hat{}\, a[i] \,\hat{}\, a[i] = H$$

所以,$(\text{sum} \,\hat{}\, a[i]) <= a[i]$就是 $H <= a[i]$。把 $a[i]$减少到 H,就是一种可行的方案。

8.5.3 图游戏与 Sprague-Grundy 函数

前面讲解的巴什游戏、尼姆游戏用 P-position 和 N-position 做分析工具,如果遇到更复杂的游戏,很难分析。有一种高级的分析方法,即 Sprague-Grundy 函数,是巴什游戏、尼姆游戏这类问题的通用方法,该方法用图作为分析工具。

图游戏的规则是:给定一个有向无环图,在一个起点上放一枚棋子,两个玩家交替将这枚棋子沿有向边进行移动,无法移动者判负。图是有向无环图的,不会有环路,保证游戏有终点。

像巴什游戏、尼姆游戏这样的 ICG 问题都可以转化为基于图的游戏。把 ICG 中的每个局势看成图上的一个结点,在每个局势和它的后继局势之间连一条有向边,就抽象成了图游戏。下面给出图游戏的严格定义。

1. 图游戏

定义:一个有向无环图 $G(X,F)$,X 是点(局势)的非空集合,F 是 X 上的函数,对于 $x \in X$,有 $F(x) \subset X$;对于给定的 $x \in X$,$F(x)$表示玩家从 x 出发能够移动到的位置;如果 $F(x)$为空,说明无法继续移动,称 x 是终点位置。

两个玩家的游戏过程按以下规则进行:一个玩家先走,起点是 x_0,然后两人交替走步;在位置 x,玩家可以选择移动到 y 点,$y \in F(x)$;位于终点位置的玩家,判负。

例如在巴什游戏中,设一次可以拿的石头是$\{1,2\}$,结点集合是 $X = \{0,1,2,\cdots,n\}$。$F(0)$为空,因为石子数量是 0,已经到达终点,无法再转移;$F(1) = \{0\}$,表示从 1 可以转移到 0;$F(2) = \{0,1\}$,表示从 2 可以转移到 0 或 1;等等。这里以 $n = 6$ 为例画出游戏图,如图 8.3 所示。

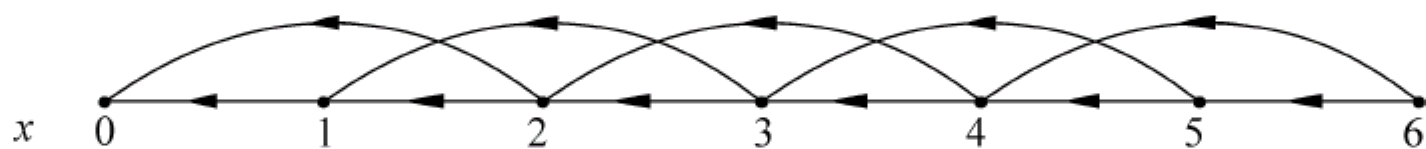


图 8.3 巴什游戏图

图 8.3 中的每个点表示一个可能的局势,箭头表示局势的转移方向。玩家的所有步骤都在这个图上。图上有一些是先手必胜点(N-position),例如 1、2、4、5 等,以及先手必败点

(P-position),例如3、6等。确定了这些关键的点,就能得到解决方案。

但是,在大部分情况下游戏图是很复杂的,例如尼姆游戏,给定3堆石头{5,7,9},图上的每个点是一个局势,如{0,0,0}、{0,1,1}等,可能的局势有$6\times8\times10=480$个,点与点之间的转移关系也很复杂。

利用Sprague-Grundy函数这个工具可以轻松地找到这些关键点。

2. Sprague-Grundy函数

定义:在一个图$G(X,F)$中,把结点x的Sprague-Grundy函数定义为sg(x),它等于没有指定给它的任意后继结点的sg值的最小非负整数。

上述定义有些拗口,下面的例子清晰地说明了它的含义。图8.3中每个结点的sg值如图8.4所示。

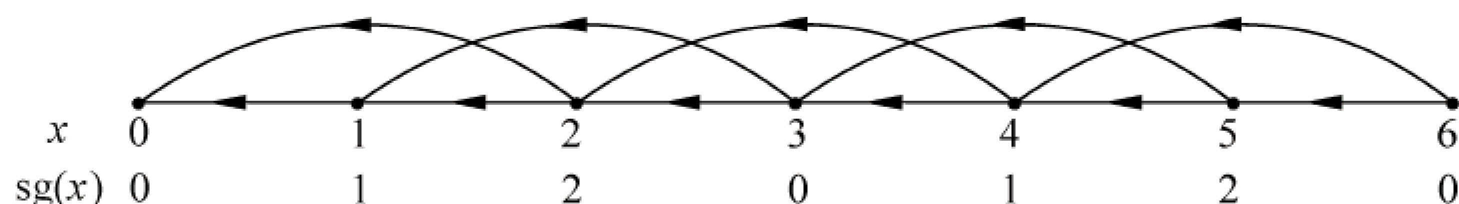


图8.4　结点x和sg(x)

当$x=0$时,sg(0)=0,因为结点0没有后继点,0是最小的非负整数;

当$x=1$时,结点1的后继是结点0,由于sg(0)=0,不等于sg(0)的最小非负整数是1,所以sg(1)=1;

当$x=2$时,结点2的后继是结点0和1,由于sg(0)=0、sg(1)=1,不等于sg(0)和sg(1)的最小非负整数是2,所以sg(2)=2;

当$x=3$时,结点3的后继是结点1和2,由于sg(1)=1、sg(2)=2,不等于sg(1)和sg(2)的最小非负整数是0,所以sg(3)=0;

当$x=4$时,结点4的后继是结点2和3,由于sg(2)=2、sg(3)=0,不等于sg(2)和sg(3)的最小非负整数是1,所以sg(4)=1;

等等。

上面的说明也给出了求每个点的sg值的过程,和前面提到的用动态规划思路求P-position、N-position的过程差不多,复杂度是$O(nm)$,其中n是石子数量,m是一次最多可拿的石子数。

3. 用Sprague-Grundy函数求解巴什游戏

在只有一堆石子的巴什游戏中,以下判断成立:

sg(x)=0的结点x是必败点,即P-position点。

证明如下:

(1) 根据sg函数的性质,有以下推论:sg(x)=0的结点x,没有sg值等于0的后继结点;sg(y)>0的任意结点y,必有一条边通向sg值为0的某个后继结点。

(2) 如果sg(x)=0的结点x是图上的终点(没有后继结点,在图论中称这个点的出度为0),显然有$x=0$,它是一个P-position点;如果x有后继结点,那么这些后续点都能通向某个sg值为0的结点。当玩家甲处于sg(x)=0的结点时,它只能转移到sg(x)≠0的结点,下一个玩家乙必然转移到sg(x)=0的点,从而再次让甲处于不利的局势。所以sg(x)=0

的点是必败点。

仍然以 hdu 1846 为例，用 Sprague-Grundy 函数的方法编程实现。

hdu 1846 程序（sg 函数）

```
#include <bits/stdc++.h>
using namespace std;
const int MAX = 1001;
int n, m, sg[MAX], s[MAX];
void getSG(){
   memset(sg, 0, sizeof(sg));
   for (int i = 1; i <= n; i++){
      memset(s, 0, sizeof(s));
      for (int j = 1; j <= m && i - j >= 0; j++)
          s[sg[i - j]] = 1;             //把 i 的后继结点放到集合 s 中
      for (int j = 0; j <= n; j++)      //计算 sg[i]
          if(!s[j]){sg[i] = j; break;}
   }
}
int main(){
   int c;  cin >> c;
   while (c--){
      cin >> n >> m;
      getSG();
      if (sg[n])   cout << "first\n";   //sg != 0,先手胜
      else         cout << "second\n";  //sg == 0,后手胜
   }
   return 0;
}
```

4. 用 Sprague-Grundy 函数求解尼姆游戏

尼姆游戏中有多堆石头，也可以用 Sprague-Grundy 函数求解。其步骤如下：

(1) 计算每一堆石头的 sg 值；

(2) 求所有石头堆的 sg 值的异或，其结论是：

若 $sg(x_1)\oplus sg(x_2)\oplus sg(x_3)\oplus\cdots\oplus sg(x_n)\neq 0$，先手必胜；

若 $sg(x_1)\oplus sg(x_2)\oplus sg(x_3)\oplus\cdots\oplus sg(x_n)=0$，先手必败。

请读者根据前面对尼姆游戏的说明以及 Sprague-Grundy 函数的特征证明其正确性。

下面用 Sprague-Grundy 函数求解 hdu 1848。

hdu 1848 "Fibonacci again and again"

两人小游戏，定义如下：一共有 3 堆石子，数量分别是 m、n、p；两人轮流走；每走一步可以选择任意一堆石子，然后取走 f 个；f 只能是菲波那契数列中的元素（即每次只能取 1、2、3、5、8 等数量）；最先取光所有石子的人为胜者。

输入：3 个整数 m、n、p($1\leqslant m,n,p\leqslant 1000$)，$m=n=p=0$ 表示输入结束。

输出：如果先手的人能赢，输出"Fibo"，否则输出"Nacci"。

这一题属于典型的尼姆游戏，程序如下：

hdu 1848 程序(sg 函数)

```
#include<bits/stdc++.h>
using namespace std;
const int MAX = 1001;
int sg[MAX], s[MAX];
int fibo[15] = {1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987};
void getSG(){                                     //计算每一堆的 sg 值
    for(int i = 0;i <= MAX;i++){
      sg[i] = i;
      memset(s, 0, sizeof(s));
      for(int j = 0; j < 15 && fibo[j]<= i; j++){
         s[sg[i - fibo[j]]] = 1;
         for(int j = 0; j <= i; j++)
            if(!s[j]) {sg[i] = j; break;}
      }
    }
}
int main(){
   getSG();                                       //预计算 sg 值
   int n,m,p;
   while(cin >> n >> m >> p && n + m + p){
      if(sg[n]^sg[m]^sg[p])  cout << "Fibo" << endl;
      else                        cout << "Nacci"<< endl;
   }
   return 0;
}
```

【习题】

hdu 1907 “John”,尼姆游戏。

hdu 2999 “Stone Game,Why are you always there?”,sg 函数。

hdu 1524 “A Chess Game”,sg 函数。

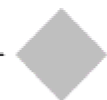
hdu 4111 “Alice and Bob”,sg 函数,记忆化搜索。

hdu 4203 “Doubloon Game”,数据规模大,找规律。

8.5.4 威佐夫游戏

威佐夫游戏(Wythoff's Game)是一种结论非常有趣的游戏,其原型见 hdu 1527 的描述。

> **hdu 1527 “取石子游戏”**
>
> 有两堆石子,数量任意,可以不同。游戏开始由两个人轮流取石子。
>
> 游戏规定每次有两种不同的取法,一是可以从任意的一堆中取走任意多的石子;二是可以从两堆中同时取走相同数量的石子。最后把石子全部取完者为胜者。
>
> 现在给出初始的两堆石子的数目 a 和 b,问先手玩家是不是最后的胜者?

分析两堆石子的数量(a,b)，使先手必输的局势有(0,0)、(1,2)、(3,5)、(4,7)、(6,10)、(8,13)、(9,15)，等等，称这些局势为“奇异局势”[①]。

观察发现，奇异局势有两个特征：①差值是递增的，分别是0,1,2,3,4,…；②每个局势的第一个值是未在前面出现过的最小的自然数。经过分析可以发现，每个奇异局势的第一个值总是等于这个局势的差值乘上黄金分割比例1.618，然后取整。

需要注意的是，在推导奇异局势时用到的黄金分割数需要较高的精度，直接用1.618这个估值是不行的。在程序中，用以下公式计算高精度黄金分割数：

```
double gold = (1 + sqrt(5))/2;
```

下面是 hdu 1527 的代码。

hdu 1527 代码

```
#include <bits/stdc++.h>
using namespace std;
int main(){
    int n, m;
    double gold = (1 + sqrt(5))/2;                  //黄金分割 = 1.618 033 98…
    while(cin >> n >> m){
        int a = min(n, m), b = max(n, m);
        double k = (double)(b - a);
        int test = (int)(k * gold);                 //乘以黄金分割数,然后取整
        if(test == a)  cout << 0 << endl;           //先手败
        else           cout << 1 << endl;           //先手胜
    }
    return 0;
}
```

8.6　小　　结

数学题是算法竞赛中的重点内容，包含的内容也相当广泛。本章讲解了一些竞赛中基本的和常用的知识点，还有很多大类没有涉及，例如积分、线性规划、傅里叶变换等。

有一些比较基础的知识点本章没有提到，但是需要读者掌握，例如高斯消元、中国剩余定理、Polya原理、欧拉函数、莫比乌斯函数等。

在一个竞赛队中，所有队员都需要掌握本章的内容，并且至少应该有一个队员深入钻研数学类题目。

① 威佐夫游戏的奇异局势和黄金分割数有关，Fibonacci数列也和黄金分割数有关，两者在这里发生了联系。关于威佐夫游戏的奇异局势和黄金分割数之间关系的证明，请参考“http://www.matrix67.com/blog/archives/6784”(永久网址：perma.cc/BCX4-9XXJ)。

第9章　字　符　串

✍常用字符串函数
✍字符串哈希
✍字典树
✍KMP
✍AC 自动机
✍后缀树和后缀数组

字符串处理是竞赛中的常见题目，除了简单的字符串查找、替换、匹配等问题以外，还有比较复杂的字符串算法，其中应用广泛的有字符串哈希、KMP、字典树(Trie Tree)、AC 自动机和后缀数组等。

9.1　字符串的基本操作

字符串的基本操作有读入、查找、替换、截取、数字和字符串转换等。下面用一个例题介绍字符串的读入、查找和替换操作。

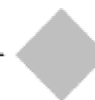
poj 3981 "字符串替换"

读取一个字符串，把其中所有的"you"替换成"we"。

下面的程序一次读取一个完整的字符串，用 gets()函数实现。

C 程序 1

```
#include <stdio.h>
char str[1002];
int main(){
  int i;
  while(gets(str) != NULL) {
    for(int i = 0; str[i]!= '\0'; i++)
        if(str[i] == 'y' && str[i + 1] == 'o' && str[i + 2] == 'u') {
           printf("we");
           i += 2;
        }
        else
           printf("%c",str[i]);
    printf("\n");
  }
```



```
  return 0;
}
```

下面的程序一次只读一个字符，用 getchar()函数实现。这个程序比上一个程序要好，因为它不需要定义一个字符串数组，当然也不用考虑数组的大小。

C 程序 2

```
#include <stdio.h>
int main(void){
    char ch1, ch2, ch3;
    while((ch1 = getchar()) != EOF) {
    if(ch1 == 'y') {
      if((ch2 = getchar()) == 'o')  {
          if((ch3 = getchar()) == 'u')
             printf("we");
          else
             printf("yo%c",ch3);
      }
      else
          printf("y%c",ch2);
    }
    else
      putchar(ch1);
}
return 0;
}
```

下面的程序用到 string 类、getline()函数。

C++ 程序

```
#include <bits/stdc++.h>
using namespace std;
int main(){
   string str;
   int pos;
   while(getline(cin, str)){
      while((pos = str.find("you")) != -1)
         str.replace(pos, 3, "we");
      cout << str << endl;
   }
   return 0;
}
```

【习题】

hdu 1062，字符串反转。

hdu 6013，字符串反转，尺取法。

hdu 5007,子串查找。

hdu 1238,求多个字符串的最大公共子串,用暴力法做。

hdu 4054,输出字符的 ASCII 码。

hdu 2055,字符串和数字转换。

hdu 5938,字符串和数字转换。

9.2 字符串哈希

首先看一个比较特殊的字符串匹配问题:在很多字符串中尽快操作某个字符串。如果字符串的规模很大,访问速度很关键,具体例子参考 hdu 2648 题。在本书第 3 章的“3.1.7 map”中曾以 hdu 2648 为例讲解了用 map 容器匹配字符串的方法。这里用字符串哈希的方法重新编程处理。

这个问题用哈希(hash)方法解决是最快的。用哈希函数对每个子串进行哈希,分别映射到不同的数字,即一个整数哈希值,然后就可以根据哈希值找到子串,接下来配合使用数据结构或 STL 完成判重、统计、查询等操作。

哈希函数是其中的核心。理论上,任意函数 $h(x)$ 都可以是哈希函数,不过一个好的哈希函数应该尽量避免冲突。这个字符串哈希函数最好是完美哈希函数。完美哈希函数是指没有冲突的哈希函数:把 n 个子串的 key 值映射到 m 个整数上,如果对任意的 key1≠key2,都有 $h(\text{key1}) \neq h(\text{key2})$,这就是完美哈希函数。此时必然有 $n \leqslant m$。更进一步,如果 $n=m$,称为最小完美哈希函数。

那么如何找到一个接近完美的字符串哈希函数?有一些经典的字符串哈希函数,例如 BKDRHash、APHash、DJBHash、JSHash 等。一般使用 BKDRHash,求得的哈希值几乎不会冲突碰撞。但在实际应用时由于得到的哈希值都很大,不能直接映射到一个巨大的空间上,所以一般需要限制空间。方法是取余:把得到的哈希值对一个设定的空间大小取余数,以余数作为索引地址。当然,这样做会产生冲突问题。

下面用字符串哈希方法重新求解 hdu 2648。


视频讲解

在下面的程序中,哈希函数 BKDRHash()计算字符串的 hash 值,返回一个 unsigned int 数。根据上面的讨论可知,由于这个数可能很大,不能直接分配空间,程序用一个较小的 N 取余,分配到大小为 N 的空间。这样做会产生冲突,所以程序的大部分代码是解决冲突问题。

hdu 2648 的字符串哈希程序

```
#include<bits/stdc++.h>
using namespace std;
const int N = 10005;
struct node{
    char name[35];
    int price;
};
vector<node> List[N];                           //用于解决冲突
```

```
unsigned int BKDRHash(char * str)  {              //哈希函数
    unsigned int seed = 31,key = 0;
    while( * str)
        key = key * seed + ( * str++);
    return key & 0x7fffffff;
}
int main(){
    int n, m, key, add, memory_price, rank, len;
    int p[N];
    char s[35];
    node t;
    while(cin >> n){
        for(int i = 0; i < N; i++)
            List[i].clear();
        for(int i = 0;i < n;i++){
            cin >> t.name;
            key = BKDRHash(t.name) % N;           //计算 hash 值,并求余
            List[key].push_back(t);               //hash 值可能冲突,把冲突的哈希值都存起来
        }
        cin >> m;
        while(m -- ){
            rank = len = 0;
            for(int i = 0; i < n; i++){
                cin >> add >> s;
                key = BKDRHash(s) % N;            //计算 hash 值
                for(int j = 0; j < List[key].size(); j++)     //处理冲突问题
                    if(strcmp(List[key][j].name, s) == 0){
                        List[key][j].price += add;
                        if(strcmp(s,"memory") == 0)
                            memory_price = List[key][j].price;
                        else
                            p[len++] = List[key][j].price;
                        break;
                    }
            }
            for(int i = 0; i < len; i++)
                if(memory_price < p[i])
                    rank++;
            cout << rank + 1 << endl;
        }
    }
    return 0;
}
```

【习题】

hdu 4821 “String”。

hdu 4080 “Stammering Aliens”。

hdu 4622 “Reincarnation”。

hdu 4622,字符串哈希,较难。

9.3 字 典 树

再次回顾一个常见的字符串匹配问题：在 n 个字符串中查找某个字符串。

如果用暴力的方法，需要逐个匹配每个字符串，复杂度是 $O(nm)$，m 是字符串的平均长度。这个操作的效率十分低。

那么有没有很快的方法？大家都有查英语字典的经验，例如查找单词"dog"，先翻到字典的 d 部分，再翻到第 2 个字母 o、第 3 个字母 g，一共找 3 次即可。查找任意单词，查找次数最多只需要这个单词的字母个数。

字典树就是模拟这个操作的数据结构，它的时间复杂度和空间复杂度都很好。

(1) 时间复杂度：插入和查找单词的复杂度都是 $O(m)$，其中 m 是待插入/查询字符串的长度。

(2) 空间复杂度：有公共前缀的单词只需要存一次公共前缀，节省了空间。

图 9.1 所示为单词 be、bee、may、man、mom、he 的字典树。

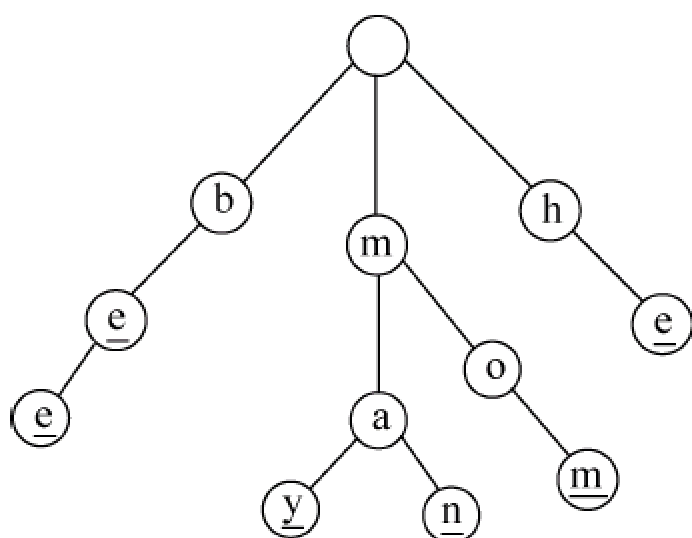


图 9.1 字典树

从图 9.1 可以归纳出字典树的基本性质：根结点不包含字符，除根结点外的每个子结点都包含一个字符；从根结点到某一个结点，路径上经过的字符连接起来，为该结点对应的字符串；每个结点的所有子结点包含的字符互不相同。

通常在实现的时候会在结点设置一个标志，标记该结点是否为单词的末尾，例如图中画线的字符。

字典树有以下常见的应用：

(1) 字符串检索。检索、查询功能是字典树的基本功能。

(2) 词频统计。统计一个单词出现了多少次。

(3) 字符串排序。在插入的时候，在树的平级按字母表的顺序插入。字典树建好之后，用先序遍历，就得到了字典树的排序。

(4) 前缀匹配。字典树是按公共前缀来建树的，很适合用于搜索提示。例如 Linux 的行命令，输入一个命令的前面几个字母，系统会自动补全命令后面的字符。

字典树在本书"9.5 AC 自动机"中也有应用。

下面的例题给出了字典树的具体实现。

> **hdu 1251 "字典树"**
>
> 很多单词只由小写字母组成，不会有重复的单词出现，统计出以某个字符串为前缀的单词数量。

该题有多种方法。

1. 用 map 实现

这一题用 map 来做非常简单，代码如下：



```
#include<bits/stdc++.h>
using namespace std;
int main(){
    char str[10];
    map<string, int> m;
    while(gets(str)){
        int len = strlen(str);
        if (!len)  break;                       //输入了一个空行
        for(int i = len; i>0; i--){
            str[i] = '\0';                      //从后往前删除这个字符串的字符,得到前缀
            m[str]++;                           //统计前缀的数量
        }
    }
    while(gets(str))  cout << m[str] << endl;
    return 0;
}
```

2. 用字典树实现

首先用正规的字典树实现，定义字典树的数据结构，并用指针指向下一层子树，代码很清晰。不过，由于本题的空间要求较高，Insert()内用 new Trie 分配的空间超过了题目的限制，代码会 MLE。

空间超额(MLE)的代码

```
#include<bits/stdc++.h>
using namespace std;
struct Trie{                                    //字典树的定义
    Trie* next[26];
    int num;                                    //以当前字符串为前缀的单词的数量
    Trie() {                                    //构造函数
       for(int i=0;i<26;i++) next[i] = NULL;
       num=0;
    }
};
Trie root;
void Insert(char str[]){                        //将字符串插入到字典树中
    Trie *p = &root;
    for(int i=0;str[i];i++){                    //遍历每一个字符
        if(p->next[str[i]-'a']==NULL)           //如果该字符没有对应的结点
           p->next[str[i]-'a'] = new Trie;      //创建一个
        p = p->next[str[i]-'a'];
        p->num++;
    }
}
int Find(char str[]){                           //返回以字符串为前缀的单词的数量
    Trie *p = &root;
    for(int i=0;str[i];i++){                    //在字典树中找到该单词的结尾位置
        if(p->next[str[i]-'a']==NULL)
           return 0;
        p = p->next[str[i]-'a'];
    }
```

```
    return p->num;
}
int main(){
    char str[11];
    while(gets(str)){
        if (!strlen(str))  break;              //输入了一个空行
        Insert(str);
    }
    while(gets(str))  cout << Find(str) << endl;
    return 0;
}
```

更好、更紧凑的存储方法是用数组来实现字典树的数据结构，在竞赛中用这种方法更加保险。相关代码如下：

用数组实现字典树

```
int trie[1000010][26];                        //用数组定义字典树，存储下一个字符的位置
int num[1000010] = {0};                       //以某一字符串为前缀的单词的数量
int pos = 1;                                  //当前新分配的存储位置
void Insert(char str[]){                      //在字典树中插入某个单词
    int p = 0;
    for(int i = 0;str[i];i++){
        int n = str[i] - 'a';
        if(trie[p][n] == 0)                   //如果对应字符还没有值
           trie[p][n] = pos++;
        p = trie[p][n];
        num[p]++;
    }
}
int Find(char str[]){                         //返回以某个字符串为前缀的单词的数量
    int p = 0;
    for(int i = 0;str[i];i++){
        int n = str[i] - 'a';
        if(trie[p][n] == 0)
           return 0;
        p = trie[p][n];
    }
    return num[p];
}
```

9.4 KMP

KMP 是单模匹配算法，即在一个长度为 n 的文本串中查找一个长度为 m 的模式串。它的复杂度是 $O(m+n)$，差不多是此类算法能达到的最优复杂度。



视频讲解

1. 朴素的模式匹配算法

在前面讲字符串哈希时曾用哈希解决了特定字符子串的匹配问题，下面

讨论更一般性的问题。

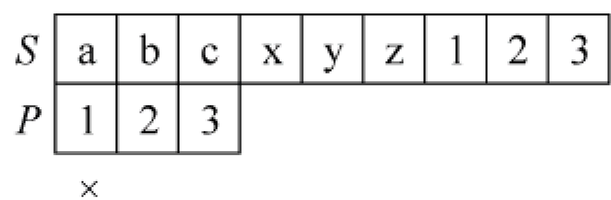
模式匹配(Pattern Matching)：在一篇长度为 n 的文本 S 中，找某个长度为 m 的关键词 P。P 可能多次出现，都需要找到。这个一般性问题用哈希算法不合适，很麻烦。

最优的模式匹配算法复杂度能达到多好？由于至少需要检索文本 S 的 n 个字符和关键词 P 的 m 个字符，所以复杂度至少是 $O(m+n)$。

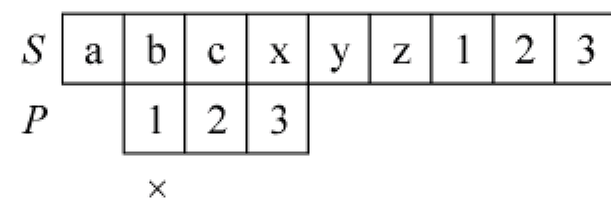
先考虑暴力方法(即朴素的模式匹配算法)：在 S 的所有字符中逐个匹配 P 的每个字符。例如，S="abcxyz123"，P="123"。第 1 次匹配，$P[0] \neq S[0]$，后面的 $P[1]$、$P[2]$就不用比较了。一共比较 6+3=9 次就好了，其中前 6 次对比 P 的第 1 个字符，第 7 次对比 P 的 3 个字符，如图 9.2 所示。

这个例子比较特殊，P 和 S 的字符基本上都不一样。在每次匹配时，往往第 1 个字符就对不上，用不着继续匹配 P 后面的字符。复杂度差不多是 $O(n+m)$，这已经是字符串匹配能达到的最优复杂度了。所以，如果字符串 S、P 符合这个特征，**用暴力法是不错的选择**。

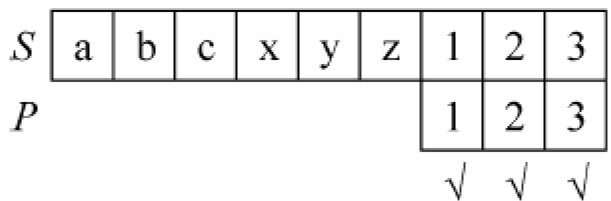
但是，如果情况比较坏，例如 P 的前 $m-1$ 个都容易找到匹配，只有最后一个不匹配，那么复杂度就退化成 $O(nm)$。例如 S= "aaaaaaaab"，P="aab"，需要尝试 6×3+3=21 次，如图 9.3 所示，远远超过上面例子中的 9 次。



(a) 第1轮匹配，首字符就失配

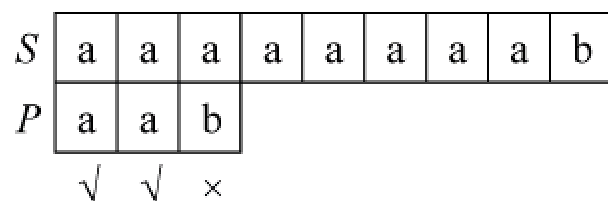


(b) 第2轮匹配，首字符就失配

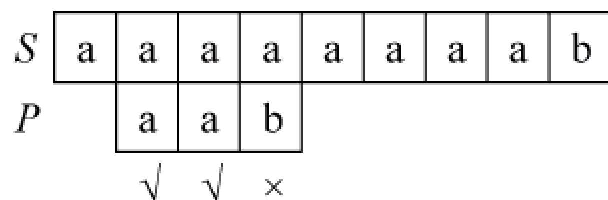


(c) 第7轮匹配，成功

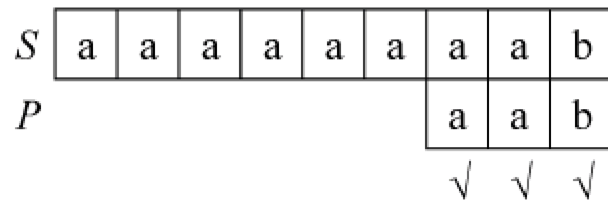
图 9.2　匹配示意



(a) 第1轮匹配，需要判断3次，不成功



(b) 第2轮匹配，也判断3次，仍不成功



(c) 第7轮匹配，成功

图 9.3　情况比较坏时的匹配

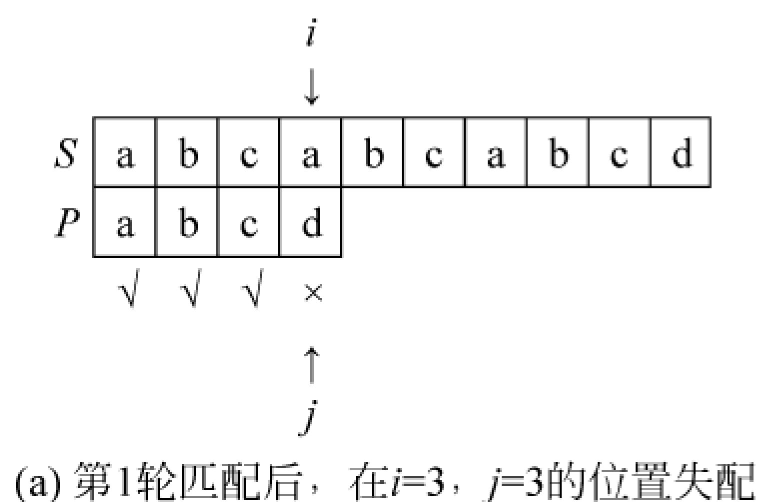
2. KMP 算法

KMP 是一种在任何情况下都能达到 $O(n+m)$复杂度的算法。它是如何做到的？简单地说，它通过分析 P 的特征对 P 进行预处理，从而在与 S 匹配的时候能够跳过一些字符串，达到快速匹配的目的。

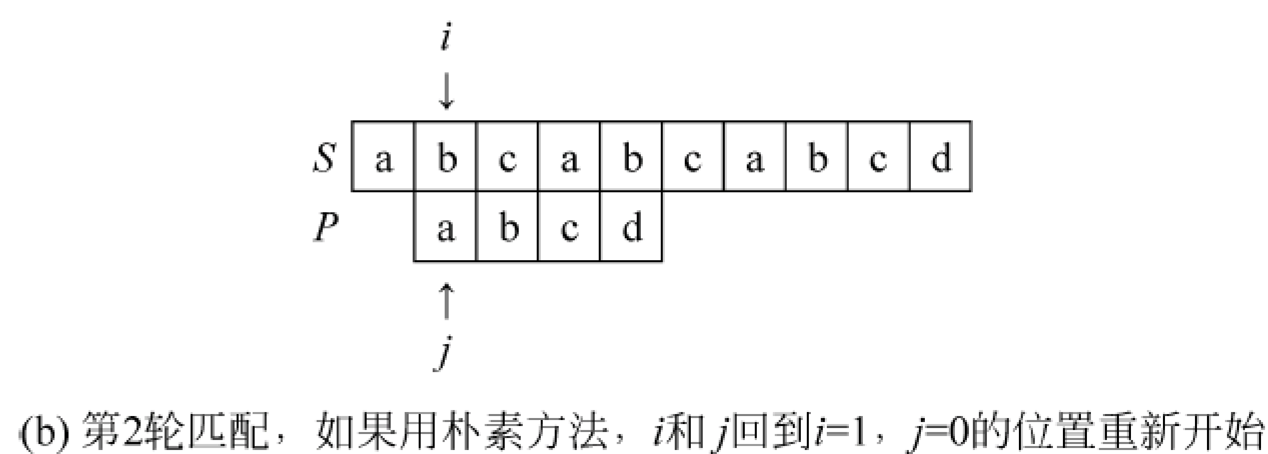
下面简单图解 KMP 的操作过程，如图 9.4 所示。$S[\,]$="abcabcabcd"，$P[\,]$="abcd"。图中的 i 指向 $S[i]$，j 指向 $P[j]$，$0 \leqslant i < n$，$0 \leqslant j < m$。

图 9.4(c)说明，在用 KMP 算法时，指向 S 的 i 指针不会回溯，而是一直往后走到底。与图 9.4(b)的朴素方法相比，大大减少了匹配次数。请读者自己分析复杂度是否为 $O(n+m)$。

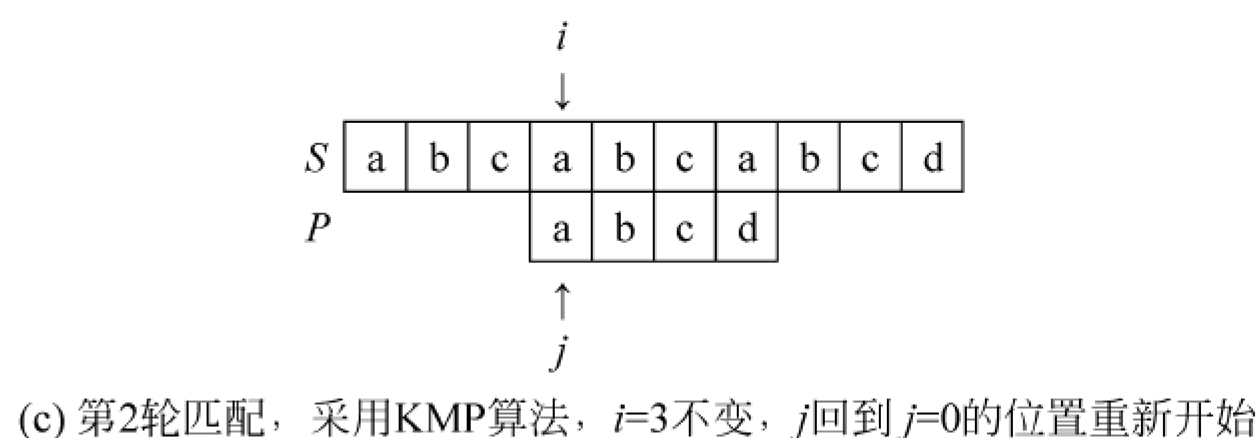
那么 KMP 是如何让 i 不回溯，只回溯 j 的呢？这就是 KMP 的核心——Next[]数组(也有写成 shift 或者 fail 的)。当出现失配后，进行下一次匹配时，用 Next[]指出 j 回溯的位置。



(a) 第1轮匹配后，在i=3，j=3的位置失配



(b) 第2轮匹配，如果用朴素方法，i和j回到i=1，j=0的位置重新开始



(c) 第2轮匹配，采用KMP算法，i=3不变，j回到j=0的位置重新开始

图 9.4　简单图解 KMP 的操作过程

Next[]是通过对 P 进行预处理得到的。在下面 hdu 2087 题的程序中用 getFail()函数求 Next[]数组。该程序虽然很短，却复杂难解，请读者自己阅读资料[①]。

有了 Next[]数组，就能很容易地写出 KMP 程序，代码见下面的例子。

3. KMP 模板题

hdu 2087 “剪花布条”

一块花布条，上面印有一些图案，另有一块直接可用的小饰条，也印有一些图案。对于给定的花布条和小饰条，计算一下能从花布条中尽可能剪出几块小饰条。

输入：每一行是成对出现的花布条和小饰条。#表示结束。

输出：输出能从花纹布中剪出的小饰条的最多个数。

输入样例：

abcde a3

aaaaaa　aa

#

输出样例：

0

3

① “从头到尾彻底理解 KMP”，网址为“https://blog.csdn.net/v_july_v/article/details/7041827（永久网址：perma.cc/FY2G-6P67）”。



本题可以完全套用 KMP 的模板。KMP 算法的模板有两部分，即 getFail()和 kmp()。getFail()预计算 Next[]数组；kmp()函数实现在 S 中找 P，注意每次匹配到的起始位置是 $s[i+1-\text{plen}]$，末尾是 $s[i]$。

找到的匹配可能有很多个，而且可能重合，例如"aaaaaa"中包含了 3 个"aa"。但在本题中需要找到能分开的子串，即剪出不同的小饰条。这个问题容易解决，只需要在程序中加一句 if(i−last >= plen)进行判断即可。

KMP 程序

```
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 1000 + 5;
char str[MAXN], pattern[MAXN];
int Next[MAXN];
int cnt;
int getFail(char *p, int plen){
              //预计算 Next[],用于在失配的情况下得到 j 回溯的位置
    Next[0] = 0; Next[1] = 0;
    for(int i = 1; i < plen; i++){
        int j = Next[i];
        while(j && p[i] != p[j])   j = Next[j];
        Next[i + 1] = (p[i] == p[j]) ? j + 1 : 0;
    }
}
int kmp(char *s, char *p) {                    //在 S 中找 P
   int last = -1;
    int slen = strlen(s), plen = strlen(p);
    getFail(p, plen);                          //预计算 Next[]数组
    int j = 0;
    for(int i = 0; i < slen; i++) {            //匹配 S 和 P 的每个字符
        while(j && s[i]!= p[j])  j = Next[j];  //失配了,用 Next[]找 j 的回溯位置
        if(s[i] == p[j])  j++;                 //当前位置的字符匹配,继续
        if(j == plen) {                        //完全匹配
            //这个匹配,在 S 中的起点是 i + 1 - plen,末尾是 i,如有需要可以打印
            //printf("at location = %d, %s\n", i + 1 - plen,&s[i + 1 - plen]);
        //-------------------下面是与本题相关的工作
        if(i - last >= plen) {                 //判断新的匹配和上一个匹配是否能分开
            cnt++;
            last = i;                          //last 指向上一次匹配的末尾位置
        }
        //-------------------
        }
    }
}
int main(){
    while(~scanf("%s", str)){                  //读串
        if(str[0] == '#')  break;
        scanf("%s", pattern);                  //读模式串
        cnt = 0;
```

```
        kmp(str, pattern);
        printf("%d\n", cnt);
    }
    return 0;
}
```

【习题】

hdu 1686/1711/2222/2896/3065/3336。

hdu 2594"Simpsons' Hidden Talents",扩展 KMP 算法,求原串 S 的每一个后缀子串与模式串 P 的最长公共前缀。

9.5 AC 自动机

AC 自动机(Aho-Corasick automaton)是 KMP 的升级版。KMP 是单模匹配算法,处理在一个文本串中查找一个模式串的问题;AC 自动机是多模匹配算法,能在一个文本串中同时查找多个不同的模式串。

多模匹配问题:给定一个长度为 n 的文本 S,以及 k 个平均长度为 m 的模式串 $P_1,P_2,\cdots,P_k$,要求搜索这些模式串出现的位置。

其实用 KMP 也能解决多模匹配问题,缺点是复杂度较高,需要对每个 $P_1,P_2,\cdots,P_k$ 分别做一次 KMP,总复杂度是 $O((n+m)k)$。

AC 自动机算法并不需要对 S 做多次 KMP,而是只搜索一遍 S,在搜索时匹配所有的模式串。

如何同时匹配所有的 P? 如果读者能结合前面介绍过的字典树就恍然大悟了。

KMP 是通过查找 P 对应的 Next[]数组实现快速匹配的。如果把所有的 P 做成一个字典树,然后在匹配的时候查找这个 P 对应的 Next[]数组,不就实现了快速匹配的效果吗?

复杂度分析:k 个模式串,平均长度为 m;文本串长度为 n。建立字典树 $O(km)$;建立 fail 指针 $O(km)$;模式匹配 $O(nm)$,乘 m 的原因是在统计的时候需要顺着链回溯到 root 结点。总时间复杂度是 $O(km+km+nm)=O(km+nm)$。

对比简单使用 KMP 的复杂度 $O((n+m)k)$,当 $m \ll k$ 时,$(k+n)m \ll (n+m)k$。AC 自动机优势非常大。

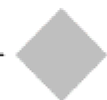
hdu 2222 题是一道模板题。

hdu 2222 "Keywords Search"

有多个关键词,在一个文本中找到它们。

输入:第 1 行是测试用例个数。每个用例包括一个整数 N,表示关键词个数,下面有 N 个关键词,$N \leqslant 10\,000$。每个关键词只包括小写字母,长度不超过 50。最后一行是文本,长度不大于 1 000 000。

输出:在输出文本中能找到多少关键词。重复的关键词只需要统计一次。



下面是代码[①]。

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 1000000 + 100;
const int SIGMA_SIZE = 26;
const int maxnode = 1000000 + 100;
int n, ans;
bool vis[maxn];
map<string, int> ms;
int ch[maxnode][SIGMA_SIZE + 5];
int val[maxnode];
int idx(char c) {return c - 'a';}
struct Trie {
    int sz;
    Trie() { sz = 1; memset(ch[0], 0, sizeof(ch[0])); memset(vis, 0, sizeof(vis)); }
    void insert(char *s) {
        int u = 0, n = strlen(s);
        for(int i = 0; i < n; i++) {
            int c = idx(s[i]);
            if(!ch[u][c]) {
                memset(ch[sz], 0, sizeof(ch[sz]));
                val[sz] = 0;
                ch[u][c] = sz++;
            }
            u = ch[u][c];
        }
        val[u]++;
    }
};
//AC 自动机
int last[maxn], f[maxn];
void print(int j) {
 if(j && !vis[j]) {
    ans += val[j]; vis[j] = 1;
    print(last[j]);
 }
}
int getFail() {
  queue<int> q;
  f[0] = 0;
  for(int c = 0; c < SIGMA_SIZE; c++) {
      int u = ch[0][c];
      if(u) {f[u] = 0; q.push(u); last[u] = 0;}
  }
  while(!q.empty()) {
      int r = q.front(); q.pop();
      for(int c = 0; c < SIGMA_SIZE; c++) {
          int u = ch[r][c];
          if(!u) {
              ch[r][c] = ch[f[r]][c];
```

① 其中，getFail()来自《算法竞赛入门经典训练指南》，刘汝佳，陈锋，清华大学出版社，3.3.3 节，214 页。

```
                continue;
            }
            q.push(u);
            int v = f[r];
            while(v && !ch[v][c]) v = f[v];
            f[u] = ch[v][c];
            last[u] = val[f[u]] ? f[u] : last[f[u]];
        }
    }
}
void find_T(char * T) {
 int n = strlen(T);
 int j = 0;
 for(int i = 0; i < n; i++) {
     int c = idx(T[i]);
     j = ch[j][c];
     if(val[j]) print(j);
     else if(last[j]) print(last[j]);
 }
}
char tmp[105];
char text[1000000 + 1000];
int main() {
 int T; cin >> T;
 while(T--) {
     scanf("%d", &n);
     Trie trie;
     ans = 0;
     for(int i = 0; i < n; i++) {
         scanf("%s", tmp);
         trie.insert(tmp);
     }
     getFail();
     scanf("%s", text);
     find_T(text);
     cout << ans << endl;
   }
     return 0;
}
```

【习题】

hdu 2243/2825/2296，AC 自动机＋DP 状态压缩。

9.6 后缀树和后缀数组

后缀树和后缀数组理解起来比较难，但是可以解决大部分字符串问题，前面提到的字符串匹配问题，例如查找子串、最长重复子串、最长公共子串等，都可以用后缀数组解决，这类题目是编程竞赛的常见题型。

本节首先讲解后缀树和后缀数组的概念，然后用后缀数组解决一些经典字符串问题。

9.6.1 概念

后缀(suffix)：一个字符串，它的一个后缀是指从某个位置开始到末尾的一个子串。例如字符串 string s＝"vamamadn"，它的后缀有 8 个，即 s[0]＝"vamamadn"、s[1]＝"amamadn"、s[2]＝"mamadn"等。具体见表 9.1 的左半部分。

表 9.1 后缀

后缀 $s[i]$	下标 i		字典序	后缀数组 sa[j]	下标 j
vamamadn	0		adn	5	0
amamadn	1		amadn	3	1
mamadn	2		amamadn	1	2
amadn	3		dn	6	3
madn	4		madn	4	4
adn	5		mamadn	2	5
dn	6		n	7	6
n	7		vamamadn	0	7

后缀树(suffix tree)：就是把所有的后缀子串用字典树的方法建立的一棵树，如图 9.5 所示。

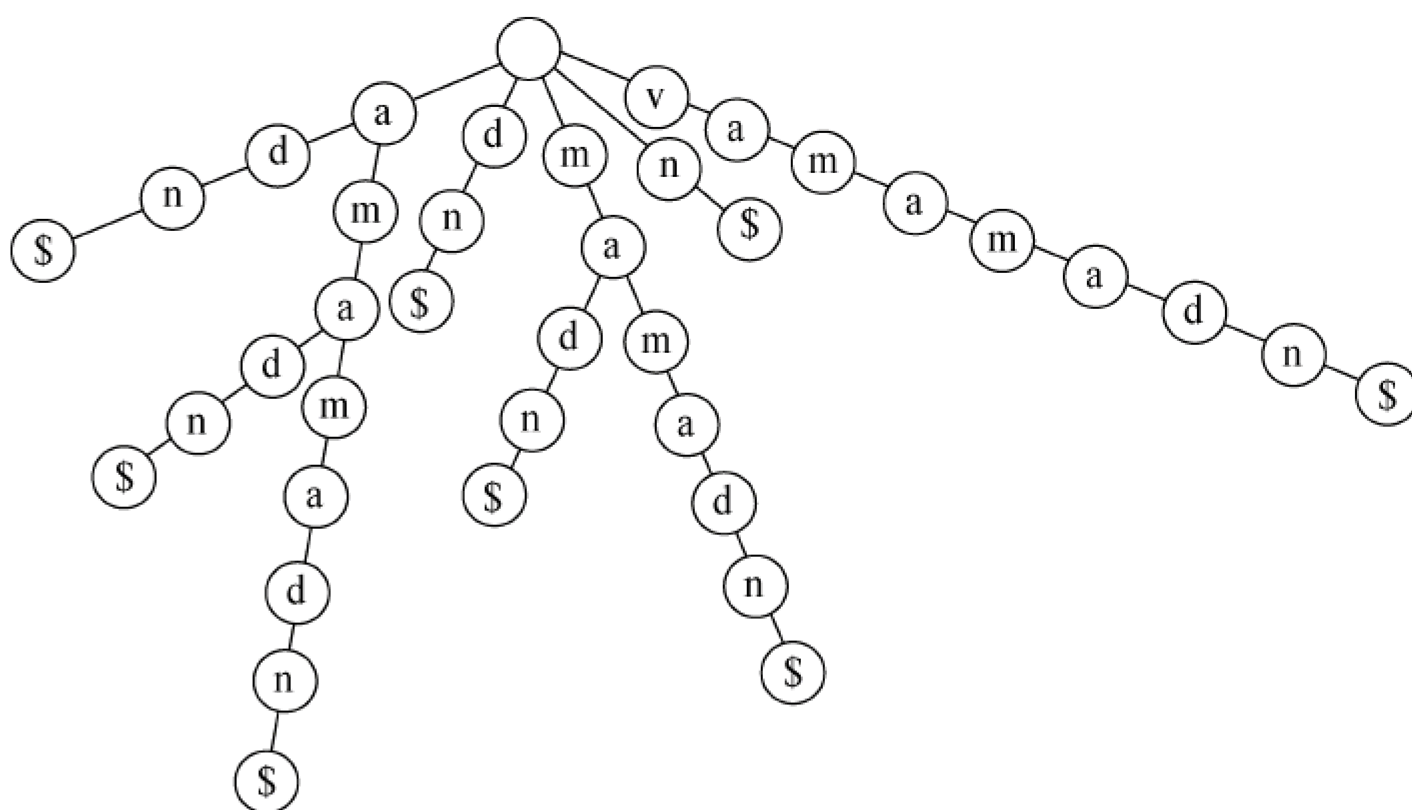


图 9.5 后缀树

其中，根结点为空，符号 $ 表示一个后缀子串的末尾。用 $ 的原因是它比较特殊，不会在字符串中出现，适合用来做标识。如果要利用后缀树查找某个子串，例如"mam"，只需要从根结点出发查 3 次即可，这就是后缀树的优势。

由于直接对后缀树进行构造和编程不太方便，所以用后缀数组(suffix array)这种简单的方法来替代。在表 9.1 中，后缀数组就是按字典序对应的后缀下标：int sa[]＝{5,3,1,6,4,2,7,0}。很明显，后缀数组的数字顺序就是后缀子串的字典顺序，记录了子串的有序排列。例如 sa[0]＝5，意思是：排名 0(即字典序最小)的子串，是原字符串中从第 5 个位置开始的后缀子串，即"adn"。

如果得到了后缀数组,可以很方便地解决一些字符串问题。下面介绍查找子串(单模匹配)问题,即在母串 s 中查找子串 t。只需要在后缀数组 sa[]上做二分搜索,就能很快地找到子串。比如查找子串 t="ad",程序如下:

```
#include <bits/stdc++.h>
using namespace std;
int find(string s, string t, int * sa){          //在 s 中查找子串 t; sa 是 s 的后缀数组
    int i = 0, j = s.length();
    while(j - i > 1) {
        int k = (i + j)/2;                          //二分法,操作 O(logn)次
        if(s.compare(sa[k], t.length(), t)< 0)     //匹配一次,复杂度是 O(m)
            i = k;
        else j = k;
    }
    if(s.compare(sa[j], t.length(), t) == 0)       //找到了,返回 t 在 s 中的位置
        return sa[j];
    if(s.compare(sa[i], t.length(), t) == 0)
        return sa[i];
    return -1;                                      //没找到
}
int main(){
    string s = "vamamadn", t = "ad";                //母串和子串
    int sa[] = {5, 3, 1, 6, 4, 2, 7, 0};            //sa[]是 s 的后缀数组,假设已经得到了
    int location = find(s, t, sa);
    cout << location <<":"<< &s[location]<< endl << endl; //打印 t 在 s 中的位置
}
```

每次查找,复杂度都是 $O(m\log_2 n)$,m 是子串长度,n 是母串长度。

在上面的程序里事先已经算好了后缀数组 sa[],所以最关键的问题是如何高效地求后缀数组?即如何对后缀子串进行排序?

常用的一种排序方法为倍增法,它的复杂度是 $O(n\log_2 n)$,下一节将详细介绍这个方法。

后缀数组是很高效的方法。例如在上面的查找子串问题中先求后缀数组,再找子串,总复杂度是 $O(n\log_2 n+m\log_2 n)$。对比经典的字符串匹配 KMP 算法,复杂度是 $O(n+m)$,前缀数组已经很接近了。如果直接用后缀树,速度更快:建树的复杂度是 $O(mn)$,在树上查找一个子串只需要比较 m 次,复杂度是 $O(m)$。

对比后缀数组和后缀树,根据前面的讲解可以知道,后缀树用空间换时间,复杂度很好;后缀数组虽然复杂度稍微差一点,但是使用的空间小,编码简单,所以在竞赛中一般使用后缀数组。

9.6.2 用倍增法求后缀数组

在讲解倍增法之前先考虑常见的排序方法,例如快速排序。快速排序,所有元素的比较次数是 $O(n\log_2 n)$,在应用到字符串排序时,每两个字符串还有 $O(n)$的比较,所以总复杂度是 $O(n^2\log_2 n)$,显然不够好。

用倍增法对后缀排序的原理比较复杂，初学者很难理解，不过如果读者按以下步骤学习就会觉得很清晰。

例：求字符串"vamamadn"的后缀数组。

第 1 步：用数字代表字母，例如 a 最小，记为 0；v 最大，记为 4。这个转换对后缀子串的排序没有影响(这一步操作实际上是对所有的后缀子串的最高位进行大小判定，不过因为很多子串的最高位相同，对应的数字也相同，所以还不能比较大小)。

第 2 步：连续两个数字的组合，相当于连续两个字符。例如 40 代表"va"、02 代表"am"等。最后一个 3 没有后续，在尾部加上 0，组成 30。这并不影响字符的比较，因为字符是从头到尾比较大小的(这一步操作是取后缀子串的最高两位，数字的大小代表子串的最高两位的大小)。

第 3 步：连续 4 个数字的组合，相当于连续 4 个字符。例如 4020 代表"vama"、0202 代表"amam"等。最后的 30 没有后续，加上 00，组成 3000(这一步操作是用数字代表后缀子串的高 4 位)。

特别需要注意的是，并没有进行连续 3 个数字的组合。原因有两个，一是不方便操作，二是并不影响后缀子串的大小比较。

在第 3 步操作后产生的 8 个数字已经全部不一样，能区分大小了。结束，并进行排序，得到 rk[]={7,2,5,1,4,0,3,6}。rk 是 rank 的缩写，表示“名次数组”。rk[]是字符串"vamamadn"的 8 个后缀子串的排序。在得到 rk[]后，可以求得后缀数组 sa[]={5,3,1,6,4,2,7,0}，如图 9.6 所示。

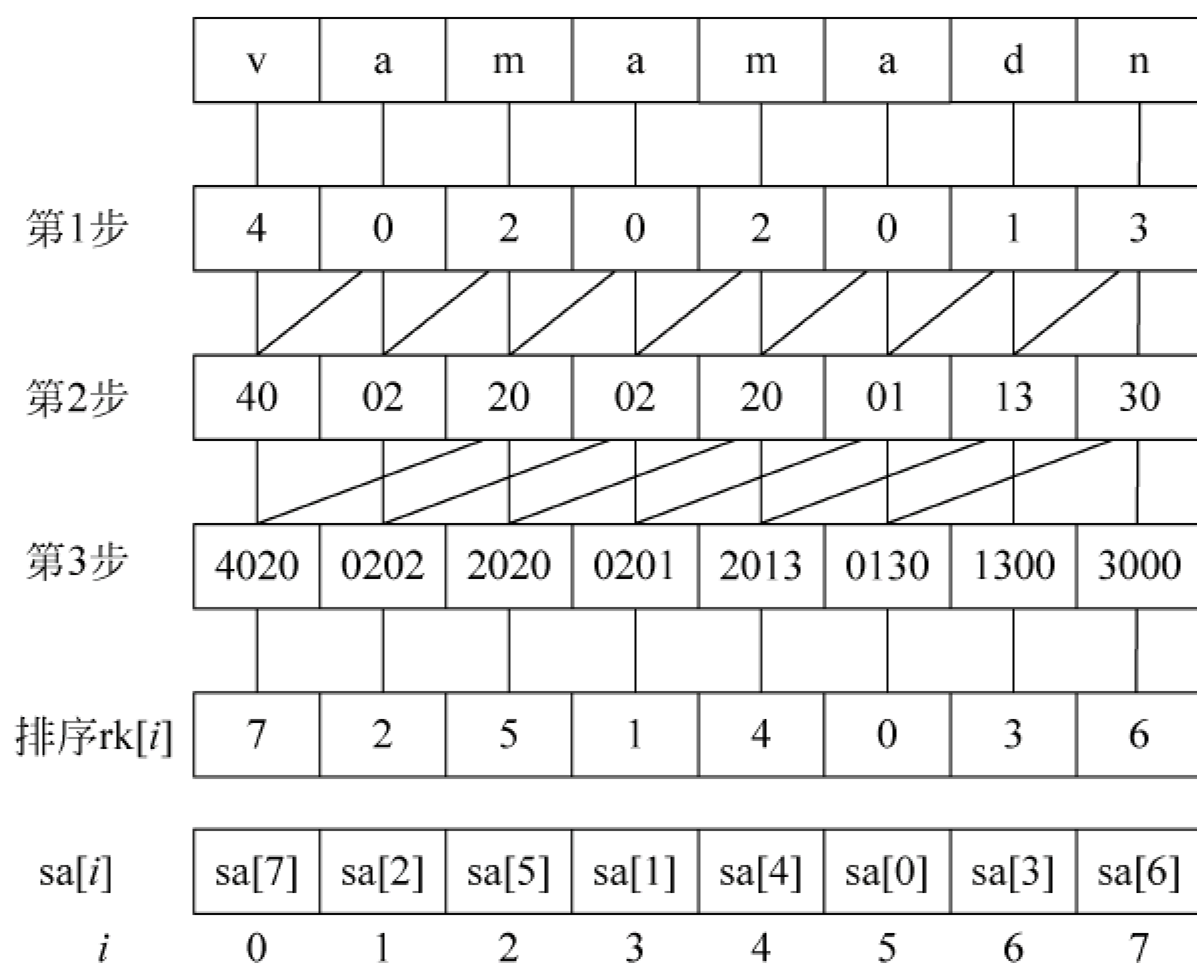


图 9.6 名次数组和后缀数组

上述操作，因为每一步都递增两倍，所以总步骤一共有 $\log(n)$步，非常少。

虽然上述过程看起来很不错，但是却并不实用。因为字符串可能很长，例如包含 1 万个字符，那么在最后一步产生的每个数字都有 10 000 位，是个天文数字，根本无法存储和排序。

那么能不能在每一步中缩小产生的组合数字的大小，而且还能保持顺序呢？答案是能。方法是在每一步操作后就对组合数字进行排序，用序号产生一个新数字，然后用新数字进

行下一步操作，过程如图 9.7 所示。

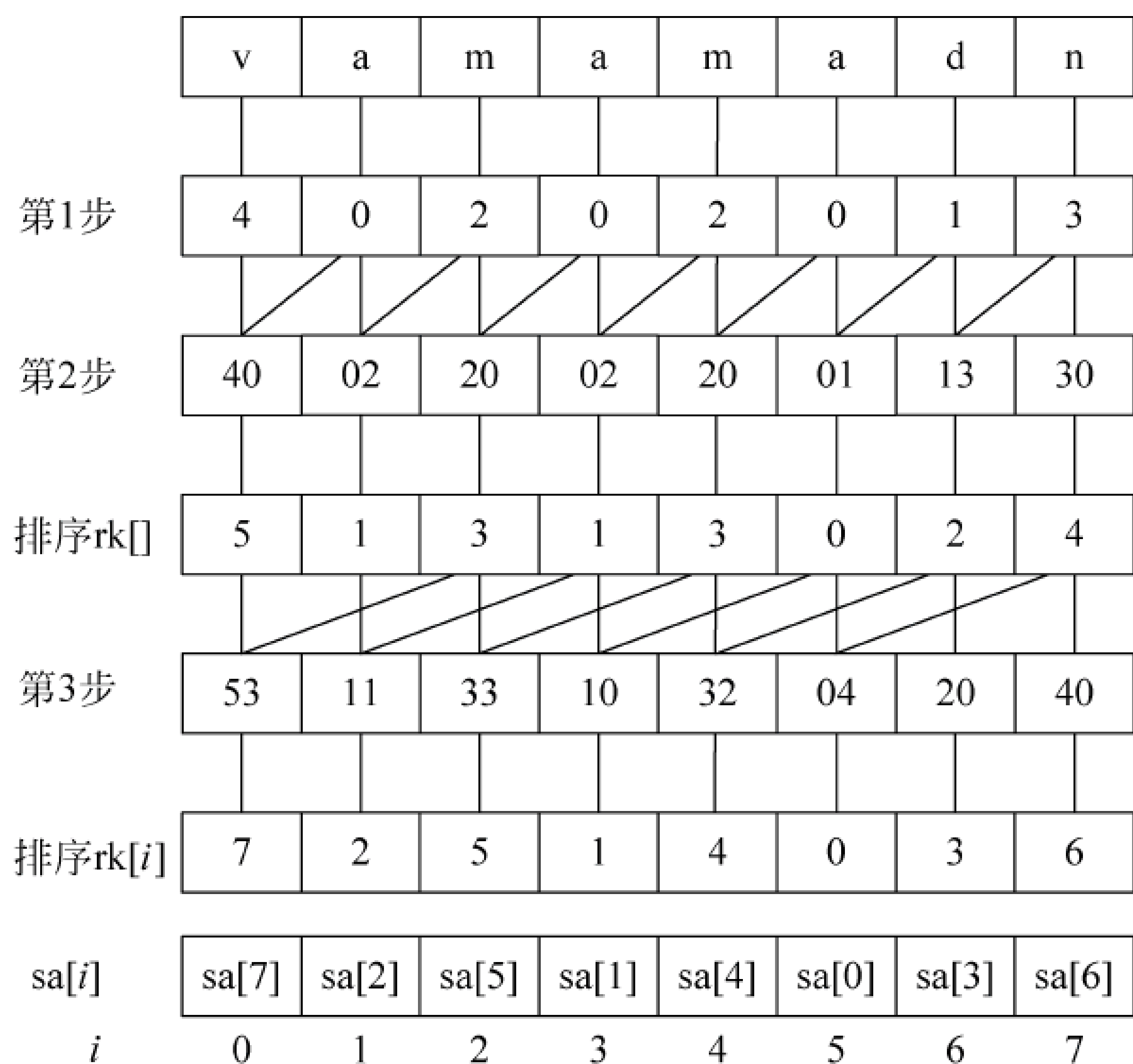


图 9.7　改进后的名次数组和后缀数组

可以发现，每一步排序后产生的新数字实际上仍然是对后缀子串的高位的排序。所以，最后的结果和图 9.6 是一样的。

产生的新数字有多大？假设字符串长度 $n=1$ 万，即每一步处理 1 万个数，那么产生的新数字是对这 1 万个数的排序结果，最大就是 10 000。所以，每一步的排序只是对 1 万个大小在 1～10 000 的数字进行排序，这是很容易做到的。

在这个过程中，核心是处理 rk[]和 sa[]。

1. sa[]、rk[]数组

在后缀数组的相关程序中，有 3 个关键的数组：sa[]、rk[]和 height[]。下面给出 sa[]、rk[]的概念和相互关系，请对照图 9.7 进行理解。

sa[]：后缀数组 suffix array。保存 $0\sim n-1$ 的全排列，含义是，把所有后缀按字典序排序后，后缀在原串中的位置。性质：suffix(sa[i])< suffix(sa[$i+1$])。sa[]记录"位置"："排第 i 的是谁？"——"排第 i 的后缀子串在原串的 sa[i]这个位置。"

rk[]：名次数组 rank array。最后得到的 rk[]也是 $0\sim n-1$ 的全排列，保存 suffix(i)在所有后缀中按字典序排序的"名次"。rk[]记录"排名"：第 i 个后缀子串排第几？"——"原串从头数第 i 个后缀子串，排名是 rk[i]。"

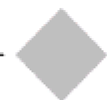
rk[]和 sa[]是一一对应关系，互为逆运算，可以互相推导：

(1) 用 rk[]推导 sa[]：

```
for(int i = 0; i < n; i++)    sa[rk[i]] = i;
```

(2) 用 sa[]推导 rk[]：

```
for(int i = 0; i < n; i++)   rk[sa[i]] = i;
```



2. 用 sort()函数求后缀数组 sa[]

下面用 STL 的 sort()函数对 rk[]排序，并求得后缀数组。程序的核心就是上面两个推导，读者需要透彻理解才能看懂下面的代码。

比较函数 comp_sa()判断每一步中得到的组合数的大小。在图 9.7 所示的原理图中，例如从第 1 步到第 2 步，把 4、0 组合成 40，把 0、2 组合成 02，等等，然后再用于比较。comp_sa()省去了组合过程，直接进行比较：首先比较 40 和 02 的高位，再比较低位。

程序的逻辑如下：

(1) 用 sort()在每一步根据当前的 rk[]计算出当前的 sa[]，请读者认真体会细节。

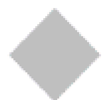
(2) 用 sa[]更新下一步用到的 rk[]。注意每一步的 sa[]，其中任意两个 sa[i]和 sa[j]都不同，但是下一步的 rk[]中有一些是相同的，所以 sa[]和 rk[]还不是一一对应的。此时需要先用 sa[]根据原来的 rk[]中的记录推导新的 rk[]，这需要用一个临时 tmp[]存放新值，然后再赋值给 rk[]。只有到了最后，sa[]和 rk[]才是一一对应的。

计算后缀数组 sa[]的模板[①]

```
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 200005;              //字符串的长度
char s[MAXN];                         //输入字符串
int sa[MAXN], rk[MAXN], tmp[MAXN + 1];
int n, k;

bool comp_sa(int i, int j){           //组合数有两个部分,高位是 rk[i],低位是 rk[i+k]
    if(rk[i] != rk[j])                //先比较高位: rk[i]和 rk[j]
        return rk[i] < rk[j];
    else{                             //高位相等,再比较低位的 rk[i+k]和 rk[j+k]
        int ri = i+k <= n? rk[i+k] : -1;
        int rj = j+k <= n? rk[j+k] : -1;
        return ri < rj;
    }
}
void calc_sa() {                      //计算字符串 s 的后缀数组
    for(int i = 0; i <= n; i++)    {
        rk[i] = s[i];                 //字符串的原始数值
        sa[i] = i;                    //后缀数组,在每一步记录当前排序后的结果
    }
    for(k = 1; k <= n; k = k*2){      //开始一步步操作,每一步递增两倍进行组合
        sort(sa, sa+n, comp_sa);      //排序,结果记录在 sa[]中
        tmp[sa[0]] = 0;
        for(int i = 0; i < n; i++)    //用 sa[]倒推组合数,并记录在 tmp[]中
          tmp[sa[i+1]] = tmp[sa[i]] + (comp_sa(sa[i],sa[i+1]) ? 1: 0);
        for(int i = 0; i < n; i++)    //把 tmp[]复制给 rk[],用于下一步操作
          rk[i] = tmp[i];
    }
```

① 参考《挑战程序设计竞赛》，秋叶拓哉，379 页，“4.7.3 后缀数组”。




```
}
int main(){
    while(scanf("%s",s)!= EOF){              //读字符串
        n = strlen(s);
        calc_sa();                           //求后缀数组 sa[]
        for(int i = 0;i < n;i++)             //打印后缀数组
            cout << sa[i]<<" ";
    }
    return 0;
}
```

上面的程序用到的 sort()实际是快速排序，每一步排序的复杂度是 $O(n\log_2 n)$，一共有 $\log_2 n$ 个步骤，总复杂度是 $O(n\log_2 n)$。虽然已经很好了，不过还有一种更快的排序方法——基数排序，总复杂度只有 $O(n\log_2 n)$。在下一节的问题 hdu 1403 中分别提交用 sort()和基数排序两种方案的倍增法程序，执行时间分别是 1000ms 和 80ms。

3. 基数排序

基数排序是一种不太符合常识的排序方法，它不是先比较高位再比较低位，而是反过来，先比较低位再比较高位。

例如排序{47,23,19,17,31}：

第 1 步：先按个位大小排序，得到{31,23,47,17,19}；

第 2 步：再按十位大小排序，得到{17,19,23,31,47}，结束，得到有序排列。

更特别的是，上述操作并不是用比较的方法得到的，而是用"哈希"的思路：直接把数字放到对应的"格子"里，第 1 步按个位放，第 2 步按十位放。表 9.2 中第 2 步得到的序列就是结果。

表 9.2 把数字放到对应的"格子"里

格　　子	0	1	2	3	4	5	6	7	8	9
第 1 步		**31**		**23**				**47、17**		**19**
第 2 步		**17、19**	**23**	**31**	**47**					

基数排序的复杂度：n 个数，每个数有 d 位(例如上面例子中的 17～47 都是两位数)，每一位有 k 种可能(十进制，0～9 共 10 种情况)，复杂度是 $O(d(n+k))$，存储空间是 $O(n+k)$。对长度 10 000 的字符串进行一次排序，$n=10\ 000$，$d\leqslant 5$，$k=10$，复杂度 $d(n+k)\leqslant 10\ 000\times 5$，而一次快排的复杂度 $n\log_2 n\approx 10\ 000\times 13$。

对比快速排序等排序方法，基数排序在 d 比较小的情况下(即所有的数字差不多大时)是更好的方法。如果 d 比较大，基数排序并不比快速排序更好。

下面的程序用基数排序求后缀数组[①]。

```
//程序的 main()部分和上面用 sort()时的一样
char s[MAXN];
int sa[MAXN],cnt[MAXN],t1[MAXN],t2[MAXN],rk[MAXN],height[MAXN];
```

① 代码改编自《算法竞赛入门经典训练指南》，刘汝佳，陈锋，清华大学出版社，3.4.1 节。

```
int n;
void calc_sa() {     //void build_sa(int n, int m)     //n 是字符串长度
   int m = 127;                              //m 是小写字母的 ASCII 码值范围.构造字符串 s 的后
                                             //缀数组,每个字符的值必须为 0～m-1
   int i, *x = t1, *y = t2;
   for(i = 0;i < m;i++)   cnt[i] = 0;
   for(i = 0;i < n;i++)   cnt[x[i] = s[i]]++;
   for(i = 1;i < m;i++)   cnt[i] += cnt[i-1];
   for(i = n-1;i >= 0;i--)  sa[--cnt[x[i]]] = i;
   //sa[]: 从 0 到 n-1
   for(int k = 1;k <= n;k = k * 2){          //利用对长度为 k 的排序的结果对长度为 2k 的排序
      int p = 0;
      //2nd
      for(i = n-k;i < n;i++)  y[p++] = i;
      for(i = 0;i < n;i++)    if(sa[i]>= k) y[p++] = sa[i] - k;
      //1st
      for(i = 0;i < m;i++)   cnt[i] = 0;
      for(i = 0;i < n;i++)   cnt[x[y[i]]]++;
      for(i = 1;i < m;i++)   cnt[i] += cnt[i-1];
      for(i = n-1;i >= 0;i--)  sa[--cnt[x[y[i]]]] = y[i];
      swap(x,y);
      p = 1; x[sa[0]] = 0;
      for(i = 1;i < n;i++)
          x[sa[i]] =
            y[sa[i-1]] == y[sa[i]]&&y[sa[i-1] + k] == y[sa[i] + k]?p-1:p++;
      if(p >= n) break;
      m = p;
   }
}
```

4. 高度数组 height[]

height[]是一个辅助数组，和最长公共前缀(Longest Common Prefix，LCP)相关。height[]数组非常重要，很多使用后缀数组解决的题目都依赖 height[]数组完成。

LCP(i,j)：suffix(sa[i])与 suffix[sa[j]]的最长公共前缀长度，即排序后第 i 个后缀和第 j 个后缀的最长公共前缀长度。

LCP(i,j)$=\min\{\text{LCP}(k-1,k)\}$，$i<k\leqslant j$

定义 height[i]为 sa[$i-1$]和 sa[i]，也就是排名相邻的两个后缀的最长公共前缀长度。例如前面的"vamamadn"中，sa[1]表示"amadn"，sa[2]表示"amamadn"，那么 height[2]=3，表示 sa[1]和 sa[2]这两个后缀的前 3 个字符相同。

用暴力的方法可以推导 height[]数组，即比较所有相邻的 sa[]，然而复杂度是 $O(n^2)$。下面给出复杂度为 $O(n)$的代码：

```
void getheight(int n){                       //n 是字符串长度
   int i, j, k = 0;
   for(i = 0 ;i < n; i++)  rk[sa[i]] = i;    //用 sa[]推导 rk[]
   for(i = 0; i < n; i++) {
      if(k)  k--;
```

```
        int j = sa[rk[i] - 1];
        while(s[i + k] == s[j + k])  k++;
        height[rk[i]] = k;
    }
}
```

height[]数组的应用非常广泛,其中最直接的应用是求最长重复子串问题、求最长公共子串问题,见下一节的讨论。

9.6.3 用后缀数组解决经典问题

在字符串问题中有这样一些经典问题,可以用后缀数组解决:

(1) 在字符串 s 中查找子串 t,具体操作见 9.6.1 节。

(2) 在字符串 s 中找最长重复子串。先求 height[]数组,其中的最大值 height[i]就是最长重复子串的长度。如果需要打印最长重复子串,它就是后缀子串 sa[$i-1$]和 sa[i]的最长公共前缀。

(3) 找字符串 s_1 和 s_2 的最长公共子串,以及扩展到求多个字符串的最长公共子串。最长公共子串(Longest Common Substring)和最长公共子序列(Longest Common Subsequence)不同。子串是串的一个连续的部分,子序列则不必连续。比如字符串"abcf"和"bcef"的最长公共子串为"bc",而最长公共子序列是"bcf"。这两个问题,在数据规模小的情况下都可以用动态规划求解,设 s_1、s_2 的长度分别是 m、n,则复杂度是 $O(mn)$。然而动态规划并不够好,如果 $m,n>10\ 000$,动态规划就不能用了,需要用后缀数组。

这个问题实际上和上一个问题"最长重复子串"类似:合并 s_1 和 s_2,得到一个大字符串 s,就变成了上一个问题。技巧是在合并的时候需要在 s_1 和 s_2 之间插入一个未出现过的特殊字符,例如'$',进行分隔,避免合并产生更长的子串。

具体操作:首先计算 height[]数组,然后查找最大的 height[i],而且它对应的 sa[$i-1$]和 sa[i]分别属于被'$'分隔的前后两个字符串时,就是解。

hdu 1403 题是最长公共子串问题。

hdu 1403 "最长公共子串"

求两个字符串的最长公共子串。

输入:每个测试用例包含两个字符串,每个字符串最多有 100 000 个字符。所有的字符都是小写的。

输出:输出最长公共子串的长度。

输入样例:

banana

cianaic

输出样例:

3

在样例中,最长公共子串是"ana",长度是 3。由于字符串长度是 100 000,程序的复杂

度不能大于 $O(n\log_2 n)$。

下面给出用后缀数组实现的程序。其中用到的 calc_sa()、getheight()函数已经在前文给出。读者可以分别用 sort()和基数排序实现的 calc_sa()提交,经验证,sort()版程序的执行时间是 1000ms,基数排序版的是 80ms。

最长公共子串程序

```
//省略了 calc_sa()、getheight()函数,已在上一节给出
int main(){
    int len1, ans;
    while(scanf("%s", s)!= EOF) {          //读第 1 个字符串
        n = strlen(s);
        len1 = n;
        s[n] = '$';                        //用'$'分隔两个字符串
        scanf("%s", s+n+1);                //读第 2 个字符串,与第 1 个合并
        n = strlen(s);
        calc_sa();                         //求后缀数组 sa[]
        getheight(n);                      //求 height[]数组
        ans = 0;
        for(int i = 1; i<n; i++)
         //找最大的 height[i],并且它对应的 sa[i-1]和 sa[i]分别属于前后两个字符串
           if(height[i]> ans &&
           ((sa[i-1]< len1 &&sa[i]>= len1) || (sa[i-1]>= len1&&sa[i]< len1)))
               ans = height[i];
        printf("%d\n",ans);
    }
    return 0;
}
```

(4) 找字符串 s 的最长回文子串。例如"helpsoshelp"的最长回文子串是"sos"。回文串一般用 Manacher 算法。

【习题】

hdu 5769,后缀数组。
hdu 3948,回文串。
hdu 4691,最长公共前缀。
hdu 5008,第 k 小子串。
hdu 4416,后缀自动机。

9.7 小　结

本章讲解的 KMP、AC 自动机、后缀数组等知识点都比较复杂,并且在应用中经常结合 DP 等其他算法。字符串算法也是算法竞赛中的难点。

第10章　图　　论

✍图的概念和存储
✍图的遍历和连通性
✍拓扑排序
✍欧拉路
✍无向图和有向图的连通性
✍2-SAT 问题
✍最短路径
✍最小生成树
✍最大流：残留网络、增广路
✍最小割
✍最小费用最大流
✍二分图匹配

图是一种很常见的模型，能描述事物或状态的关系，很多问题可以抽象为图论问题。图论的算法十分丰富，常见的问题或算法有 60 多个。在算法竞赛中，图论属于比较难的内容。

本章讲解图论的基本概念、图论常用的数据结构、常见的图论、网络流算法，并通过经典题目分析建模过程，给出标准程序。

10.1　图的基本概念

图是常见的抽象模型，由点(node，或者 vertex)和连接点的边(edge)组成。图是点和边构成的网。图描述了事物之间的连接。图最典型的应用场景是地图，地图由地点和道路组成，它的特征如下。

(1) 地点：可能是十字路口，也可能是三岔路口，或者仅仅是一个连接点。在图论中，把地点抽象为点。

(2) 道路：可能是单行道或双行道。抽象成有向边或无向边。

(3) 道路有过路费：抽象成边的权值。

(4) 求两点间的最短道路，即图论里的最短路径算法。

(5) 在城市群之间如何修最短的连通道路，即图论中的最小生成树问题。

地图的这些问题都是图论研究的对象。

计算机网络也是典型的图问题，和地图非常相似。

人际关系也可以抽象成图，即社交网络。例如著名的“六度空间理论”，世界上任意两个人，最多通过 5 个中间人就能联系到。把人看成点，把人和人之间的关系看成边，这就是一

个图的连通性问题。

树，即连通无环图，它是一种特殊的图。树的结点从根开始，层层扩展子树，是一种层次关系，这种层次关系保证了树上的结点不会出现环路。在图的算法中，经常需要在图上生成一棵树，再进行操作。

根据边有无方向、有无权值、有无环路，可以把图分成很多种，例如：

(1) 无向无权图，边没有权值、没有方向；

(2) 有向无权图，边有方向、无权值；

(3) 加权无向图，边有权值，但没有方向；

(4) 加权有向图；

(5) 有向无环图(Directed Acyclic Graph，DAG)。

图算法的复杂度显然和边的数量 E、点的数量 V 相关。如果一个算法的复杂度是线性时间 $O(V+E)$，这几乎是图问题中能达到的最好程度了。如果能达到 $O(V\log_2 E)$、$O(E\log_2 V)$或类似的复杂度，则是很好的算法。如果是 $O(V^2)$、$O(E^2)$或更高，在图问题中不算是好的算法。

10.2 图的存储

对图的任何操作都需要基于一个存储好的图。图的存储结构必须是一种有序的存储，能让程序很快定位结点 u 和 v 的边(u,v)，最好能在 $O(1)$的时间内只用一次或几次就定位到。

一般用 3 种数据结构存储图，即邻接矩阵、邻接表、链式前向星。

以图 10.1 所示的有向图为例，图中有 6 个结点、11 条边。

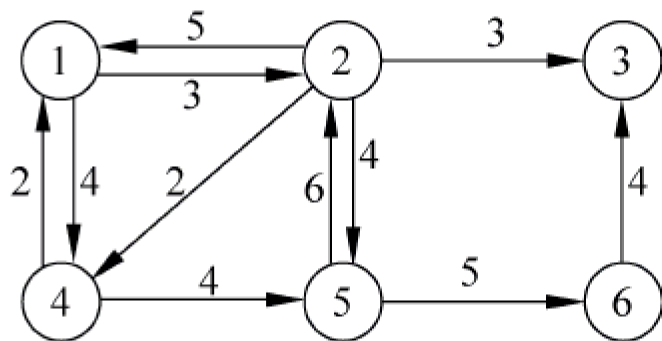


图 10.1 有向图

1. 邻接矩阵

用二维数组存储即可：int graph[NUM][NUM]。

无向图：graph[i][j]=graph[j][i]。

有向图：graph[i][j]!=graph[j][i]。

权值：graph[i][j]存结点 i 到 j 的边的权值，例如 graph[1][2]=3、graph[2][1]=5 等。用 graph[i][j]=INF 表示 i 和 j 之间无边。

优点：适合稠密图；编码非常简短；对边的存储、查询、更新等操作又快又简单，只需要一步就能访问和修改。

缺点：

(1) 存储复杂度 $O(V^2)$太高。如果用来存稀疏图，大量空间会被浪费。例如上面的图，6 个点，只有 11 条边，但是 graph[6][6]的空间是 36。当 $V=10\ 000$ 个结点时，空间为 100MB，已经超过了常见 ACM 竞赛题的空间限制，而一百万个点的图在 ACM 题中是很常见的。

(2) 一般情况下不能存储重边。(u,v)之间可能有两条或更多的边，这些边的费用不同、容量不同，是不能合并的。有向边(u,v)在矩阵中只能存储一个参数，矩阵本身的局限

性使它不能存储重边。不过,如果这个参数值只是用来表示边的数量,也算是存储了重边。

2. 邻接表

邻接表的概念请阅读《数据结构》教材,规模大的稀疏图一般用邻接表存储。它的优点是存储效率非常高,只需要与边数成正比的空间,存储复杂度为 $O(V+E)$,几乎已经达到了最优的复杂度,而且能存储重边;缺点是编程比邻接矩阵麻烦一些,访问和修改也慢一些。

在本章“10.9.3 SPFA”这一节中用 STL 的 vector 实现了邻接表,有关代码如下:

```
//定义边
    struct edge{
        int from, to, w;                                  //边:起点 from,终点 to,权值 w
        edge(int a, int b,int c){from = a; to = b; w = c;} //对边赋值
    };
    vector < edge > e[NUM];                               //e[i]: 存第 i 个结点连接的所有边
//初始化
    for(int i = 1; i <= n; i++)
        e[i].clear();
//存边
     e[a].push_back(edge(a,b,c));                         //把边(a,b)存到结点 a 的邻接表中
//检索结点 u 的所有邻居
     for(int i = 0; i < e[u].size(); i++){               //结点 u 的邻居有 e[u].size()个
        …
     }
```

例如,在上面的图 10.1 中,结点 2 的邻接表是(2 1 5)→(2 3 3)→(2 4 2)→(2 5 4)。

3. 链式前向星

用邻接表存图非常节省空间,一般的大图也够用了。然而,如果空间极其紧张,有没有更紧凑的存图方法呢?邻接表有没有改进的空间?

分析邻接表的组成,存储一个结点 u 的邻接边,其方法的关键是先定位第 1 个边,第 1 个边再指向第 2 个边,第 2 个边再指向第 3 个边,依此类推,根据这个分析,可以设计一种极为紧凑、没有任何空间浪费、编码非常简单的存图方法。图 10.2 是对前面的图 10.1 生成的存储空间,其中,head[NUM]是一个静态数组,struct edge 是一个结构的静态数组。

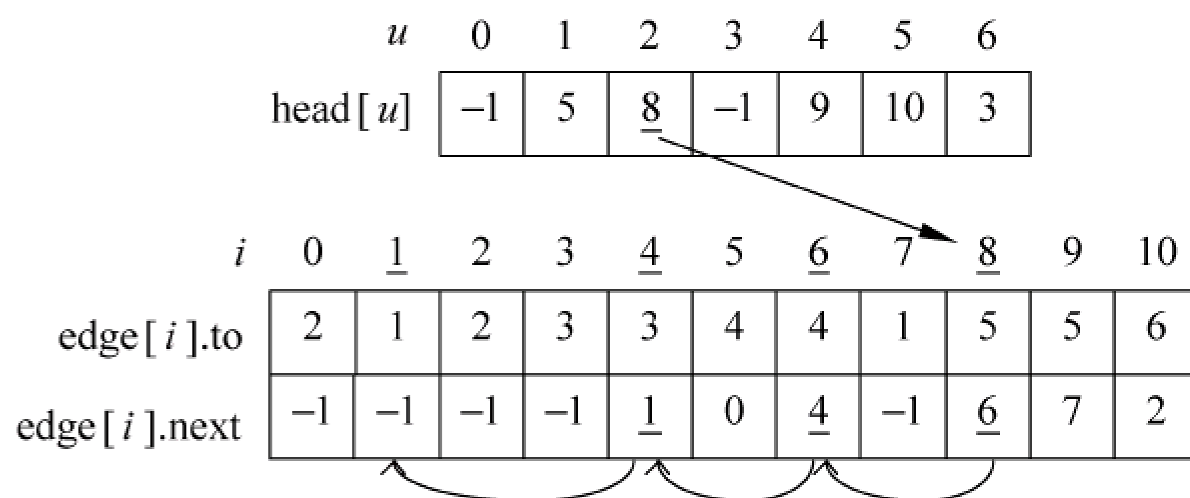
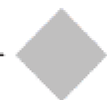


图 10.2 链式前向星存图

以结点 2 为例,从点 2 出发的边有 4 条,即(2,1)、(2,3)、(2,4)、(2,5),邻居是 1、3、4、5。

(1) 定位第 1 个边。用 head[]数组实现,例如 head[2]指向结点 2 的第 1 个边,head[2]=8,它存储在 edge[8]这个位置。



(2) 定位其他边。用 struct edge 的 next 参数指向下一个边。edge[8]. next＝6，指向下一个边在 edge[6]这个位置，然后 edge[6]. next＝4，edge[4]. next＝1，最后 edge[1]. next＝－1，－1 表示结束。

struct edge 的 to 参数记录这个边的邻居结点。例如 edge[8]. to＝5，第一个邻居是点 5；然后 edge[6]. to＝4，edge[4]. to＝3，edge[1]. to＝1，得到邻居是 1、3、4、5。

上述存储方法被称为**"链式前向星"**，它是空间效率最高的存储方法，因为它用静态数组模拟邻接表，没有任何浪费。

那么如何生成上述的存储结果？下面的程序片段来自后面 SPFA 这一节的例子。每执行一次 addedge()，就把一个新的边存入空间。

按以下顺序处理图中所有的边(u,v)：(1,2)、(2,1)、(5,2)、(6,3)、(2,3)、(1,4)、(2,4)、(4,1)、(2,5)、(4,5)、(5,6)，得到图 10.2。输入的顺序会影响结果。

从执行过程可知，每加入一个新的边，都是直接加在整个 edge[]的末尾，而与这个边的特征毫无关系。

下面是程序。

```
const int NUM = 1000005;              //一百万个点,一百万个边
struct Edge{
  int to, next, w;                    //边: 终点 to、权值 w、下一个边 next.起点放在 head[]中
}edge[NUM];
int head[NUM];                        //head[u]指向结点 u 的第一个边的存储位置
int cnt;                              //记录 edge[]的末尾位置,新加入的边放在末尾
void init(){                          //初始化
   for(int i = 0; i < NUM; ++i){
      edge[i].next = -1;              //-1: 结束,没有下一个边
      head[i] = -1;                   //-1: 不存在从结点 i 出发的边
   }
   cnt = 0;
}
void addedge(int u, int v, int w){
  edge[cnt].to = v;
  edge[cnt].w = w;
  edge[cnt].next = head[u];           //指向结点 u 上一次存的边的位置
  head[u] = cnt++;                    //更新结点 u 最新边的存放位置: 就是 edge 的末尾
}
//遍历结点 i 的所有邻居
for(int i = head[u]; ~i; i = edge[i].next)  //~i 也可以写成 i!= -1
{   …    }
```

链式前向星的优点是存储效率高、程序简单、能存储重边；缺点是不方便做删除操作。作为练习，读者可以自己编写删除的程序。

链式前向星的例程见 10.9 节。

10.3　图的遍历和连通性

图的基本特征是点和边，图的基本算法是用搜索来处理点和边的关系。第 4 章介绍了用 BFS 和 DFS 遍历一个图，在遍历的同时也解决了图的连通性问题。BFS 和 DFS 是图论

的基本算法，本章大部分内容是基于它们的。这些算法，或者直接用 BFS 和 DFS 来解决问题，或者用其思想建立新的算法。请读者回顾 BFS 和 DFS 的内容，透彻理解并能熟练写出程序。

特别是 DFS，用递归来搜索图，比 BFS 更难理解；但是一旦理解之后，编程将十分便利。图论中的很多算法，例如拓扑排序、强连通分量等，都建立在 DFS 之上。

下面是 DFS 的示例程序，其中用 vector 邻接表来存图。用矩阵存图的 DFS 示例见第 4 章。

```
vector<int> G[N];                          //G[u][i]：第 u 个结点直连的第 i 个结点
int vis[N];                                //点的访问标志，vis = 0 表示未访问过
                                           //vis = 1 表示已经被正常处理过
                                           //vis = -1 表示正在被访问中，这在有些判断中有用
                                           //例如在拓扑排序中，用于判断跳出死循环
bool dfs(int u) {                          //以 u 为起点开始 DFS 搜索
    vis[u] = 1;                            //在本次递归中被正常访问
    {… ; return true;}                     //出现目标状态，正确返回
    {… ; return false;}                    //做相应处理，返回错误
    for(int i = 0; i < G[u].size(); i++ ) { //u 的邻居有 G[u].size()个
        int v = G[u][i];                   //v 是第 i 个邻居
        if(!vis[v])                        //如果 v 没有访问过
            return dfs(v);                 //递归访问第 v 个邻居
    }
    {… ; }                                 //事后处理，返回正确或错误
}
```

下面用图 10.3 所示的例子来帮助读者理解 DFS 在图中的应用。这个例子故意设计成非连通图，所以从一个点出发并不能访问到所有点。

1. 求某个点的连通性

对需要的点执行 dfs()，就能找到它连通的点。例如找图 10.3 中 e 点的连通性，执行 dfs(e)，访问过程见图 10.4 结点上面的数字，顺序是 *ebdca*。

递归返回的结果见结点下面画线的数字，顺序是 *acdbe*。虚线指向的结点表示不再访问，因为前面已经被访问过。

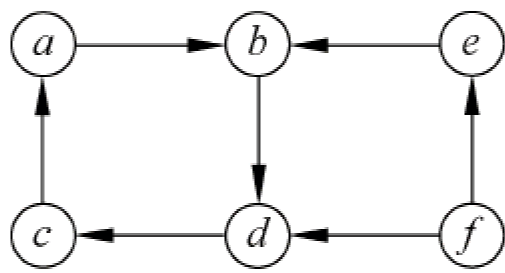


图 10.3　一个有向图例子

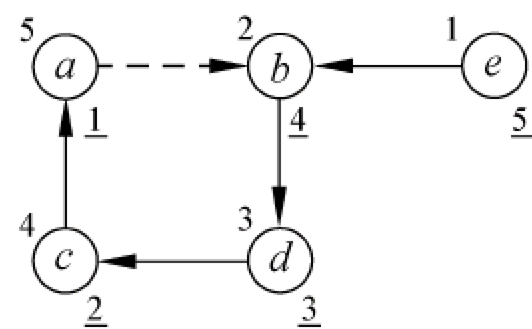


图 10.4　dfs()的访问顺序

2. 重要概念

深搜优先生成树：上面 DFS 的结果生成了一棵树，称为深搜优先生成树(depth-first spanning tree)。

树边：树上的边称为树边(tree edge)。

回退边：虚线表示的边(a,b)称为回退边(back edge)，它不在树上。

在这棵树上,从起点到其他任何一个点**只有**一条路径。如果是无向图生成的树,那么任意两个点之间只有一条路径。

3. 用 dfs()处理所有点

题目经常需要处理所有的点,也可以用 dfs()实现。其思路是想象有一个虚拟结点 v,它连接了所有的点,那么在主程序中这样进行 dfs():

```
for(int i = 0; i < n; i++)
    dfs(i);
```

读者先自己思考,写出对图 10.3 做 dfs()的过程,然后与下面的答案对照。



视频讲解

按字母顺序执行 dfs(),访问过程见图 10.5 结点上面的数字,顺序是 $abdcefghi$。虚线指向的结点表示不再访问。

递归返回的结果见结点下面画线的数字,顺序是 $cdbaefhgi$。

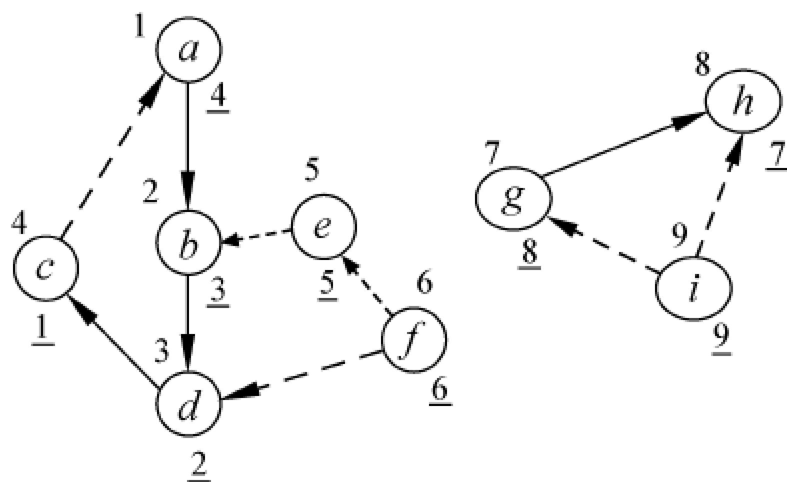


图 10.5　dfs()访问所有点的顺序

请读者彻底掌握本节的内容,这是本章后面内容的基础。

10.4　拓扑排序

BFS 和 DFS 的一个直接应用是拓扑排序。

在现实生活中,人们经常要做一连串事情,这些事情之间有顺序关系或者有依赖关系,在做一件事情之前必须先做另一件事,比如安排客人的座位、穿衣服的先后、课程学习的先后等。这些事情都可以抽象为图论中的拓扑排序。

1. 拓扑排序的概念

设有 a、b、c、d 等事情,其中 a 有最高优先级,b、c 优先级相同,d 是最低优先级,表示为 $a \rightarrow (b,c) \rightarrow d$,那么 $abcd$ 或 $acbd$ 都是可行的排序。把事情看成图的点,把先后关系看成有向边,问题转化为在图中求一个有先后关系的排序,这就是拓扑排序,如图 10.6 所示。

显然,一个图能进行拓扑排序的充要条件是它是一个有向无环图(DAG)。有环图不能进行拓扑排序。

2. 图的入度和出度

拓扑排序需要用到点的入度和出度的概念。

出度:以点 u 为起点的边的数量称为 u 的出度。

入度:以点 v 为终点的边的数量称为 v 的入度。

一个点的入度和出度体现了这个点的先后关系。如果一个点的入度等于 0,则说明它是起点,是排在最前面的;如果它的出度等于 0,则说明它是排在最后面的。例如在图 10.7 中,点 a、c 的入度为 0,它们都是优先级最高的事情;d 的出度为 0,它的优先级最低。

拓扑排序可以有多个，例如图 10.7 中的 a 和 c，谁排在前面都可以，b 和 c 也是。

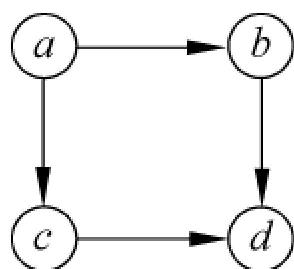


图 10.6　用图表示先后关系

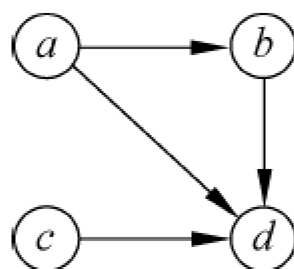


图 10.7　入度和出度

拓扑排序用 BFS 或者 DFS 都能实现。

3. 基于 BFS 的拓扑排序

基于 BFS 的拓扑排序有两种思路，即无前驱的顶点优先、无后继的顶点优先。

下面先讲解无前驱的顶点优先拓扑排序。其方法是先输出出度为 0（无前驱，优先级最高）的点，具体操作如图 10.8 所示，其中 Q 是 BFS 的队列：

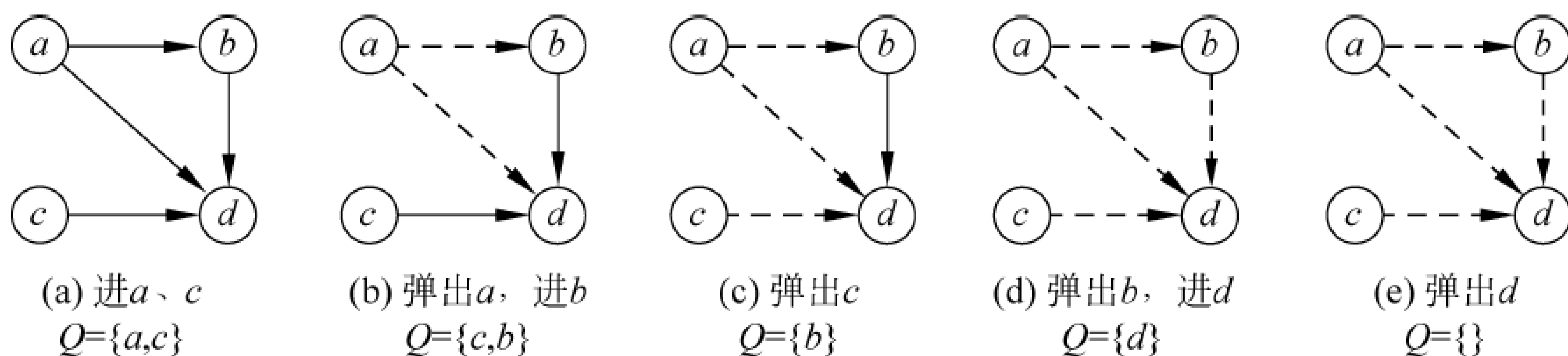


图 10.8　无前驱的顶点优先拓扑排序

步骤简述如下：

（1）找到所有入度为 0 的点，放进队列，作为起点，这些点谁先谁后没有关系。如果找不到入度为 0 的点，说明这个图不是 DAG，不存在拓扑排序。图 10.8(a) 中 a、c 的入度为 0，进队列。

（2）弹出队首 a，a 的所有邻居点，入度减 1，把入度减为 0 的邻居点 b 放进队列，没有减为 0 的点不能放进队列。内容见图 10.8(b)。

（3）继续上述操作，直到队列为空。内容见图 10.8(c)、(d)、(e)。

队列输出 $acbd$，而且包含了所有的点，这就是一个拓扑排序。

拓扑排序无解的判断：如果队列已空，但是还有点未进入队列，那么这些点的入度都不是 0，说明图不是 DAG，不存在拓扑排序。

以上是"无前驱"的思路。读者很容易发现，这个过程可以反过来执行，即"无后继的顶点优先"：从出度为 0（无后继，优先级最低）的点开始，逐步倒推。其示意图如图 10.9 所示，请读者自己分析过程。最后输出的是逆序 $dbca$。

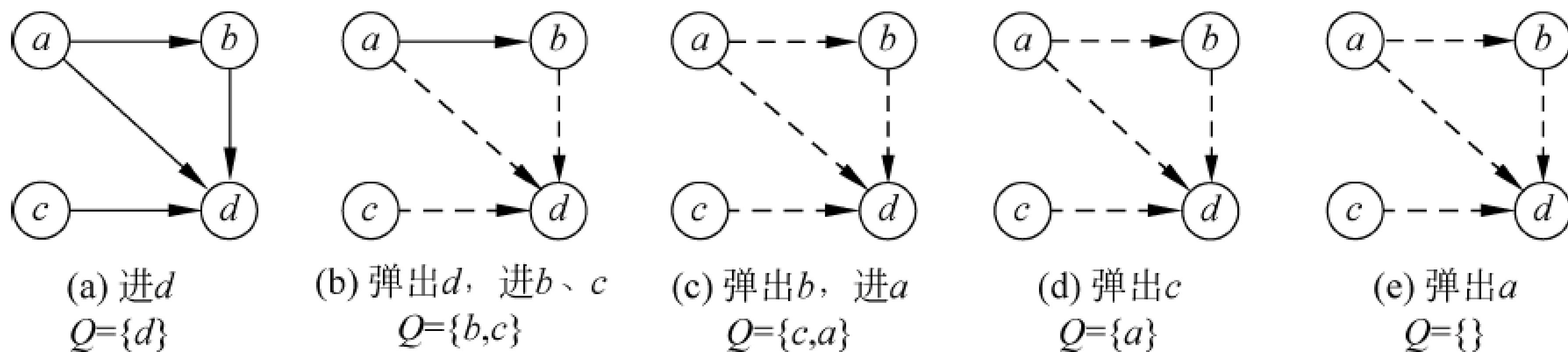


图 10.9　无后继的顶点优先拓扑排序

复杂度分析。在初始化时，查找入度为 0 的点，需要检查每个边，复杂度为 $O(E)$；在队列操作中，每个点进出队列一次，需要检查它直接连接的所有邻居，复杂度是 $O(V+E)$。其总复杂度是 $O(V+E)$。

4. 基于 DFS 搜索的拓扑排序

DFS 天然适合拓扑排序。

回顾 DFS 深度搜索的原理，是沿着一条路径一直搜索到最底层，然后逐层回退。这个过程正好体现了点和点的先后关系，天然符合拓扑排序的原理。事实上，在 DFS 上加一点点处理就能解决拓扑排序问题。

一个有向无环 DAG 图，如果只有一个点 u 是 0 入度的，那么从 u 开始 DFS，DFS 递归**返回**的顺序就是拓扑排序(是一个**逆序**)。DFS 递归返回的首先是最底层的点，它一定是 0 出度点，没有后续点，是拓扑排序的最后一个点；然后逐步回退，最后输出的是起点 u；输出的顺序是一个**逆序**。

以图 10.10 为例，从 a 开始，递归返回的顺序见点旁边画线的数字，即 $cdba$，是拓扑排序的逆序。

为了按正确的顺序打印出拓扑排序，编程时的处理是定义一个拓扑排序队列 list，每次递归输出的时候把它插到当前 list 的最前面，最后从头到尾打印 list，就是拓扑排序。这实际上是一个**栈，直接用** STL 的 stack < int >定义栈也行。

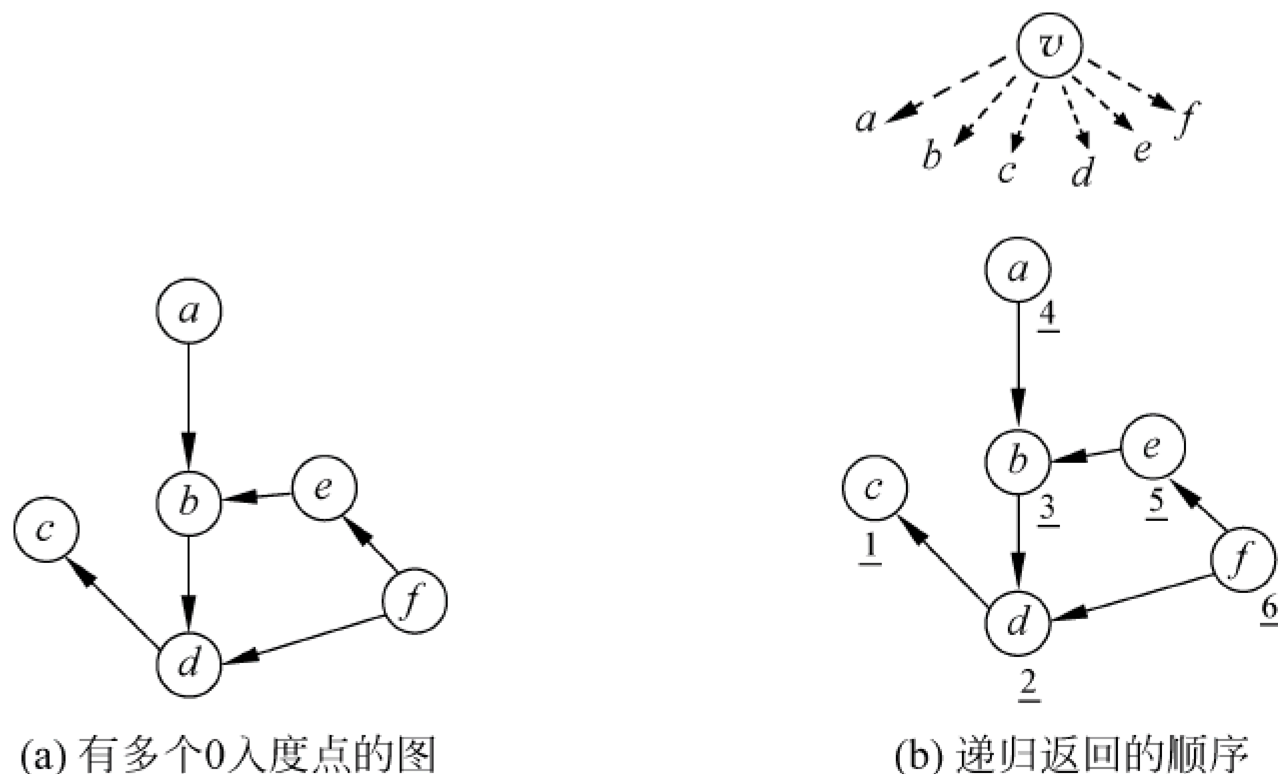
图 10.10　递归和拓扑排序

读者可以自己画个 DAG 图，体会 DFS 和拓扑排序的关系。

但是还有一些细节需要处理。

(1) 应该以入度为 0 的点为起点开始 DFS。如何找到它？需要找到它吗？如果有多个入度为 0 的点呢？

这几个问题其实并不用特别处理。10.3 节已介绍了这个做法：**想象**有一个虚拟的点 v，它单向连接到所有其他点。这个点就是图中唯一的 0 入度点，图中所有其他的点都是它的下一层递归，而且它不会把原图变成环路。从这个虚拟点开始 DFS 就完成了拓扑排序。例如图 10.11(a)有两个 0 入度点 a 和 f；图 10.11(b)想象有个虚拟点 v，那么递归返回的顺序见点旁边画线的数字，返回的是拓扑排序的逆序。



(a) 有多个0入度点的图　　(b) 递归返回的顺序

图 10.11　有多个 0 入度点的图及递归返回的顺序

在实际编程的时候并不需要处理这个虚拟点，只要在主程序中把每个点轮流执行一遍DFS即可。这样做相当于显式地递归了虚拟点的所有下一层点。

（2）如果图不是DAG，能判断吗？

图不是DAG，说明图是有环图，不存在拓扑排序。那么在递归的时候会出现**回退边**。如果读者不理解这一点，请回顾上一节的内容。

在程序中这样发现回退边：记录每个点的状态，如果dfs()递归到某个点时发现它仍在前面的递归中没有处理完毕，说明存在回退边，不存在拓扑排序。

5. 输出字典序最小的拓扑排序

由于一个图的拓扑排序有很多，题目一般不会要求输出所有的，而是输出字典序最小的那一个。

> **hdu 1285“确定比赛名次”**
>
> 有 N 个比赛队进行比赛，编号依次为 $1,2,\cdots,N$，$1\leqslant N\leqslant 500$。比赛结束后，只知道每场比赛的结果。请编程确定排名。由于可能有多种结果，输出按队伍编号排序最小的那个排名。

思路很简单：在当前步骤，在所有入度为0的点中输出编号最小的。

先考虑用BFS实现。

修改BFS的拓扑排序程序，把普通队列改为**优先队列** Q。在 Q 中放进入度为0的点，每次输出编号最小的结点，然后把它的后续结点的入度减一，入度减为0的再放进 Q。这样就能输出一个字典序最小的拓扑排序。图10.12是示例。

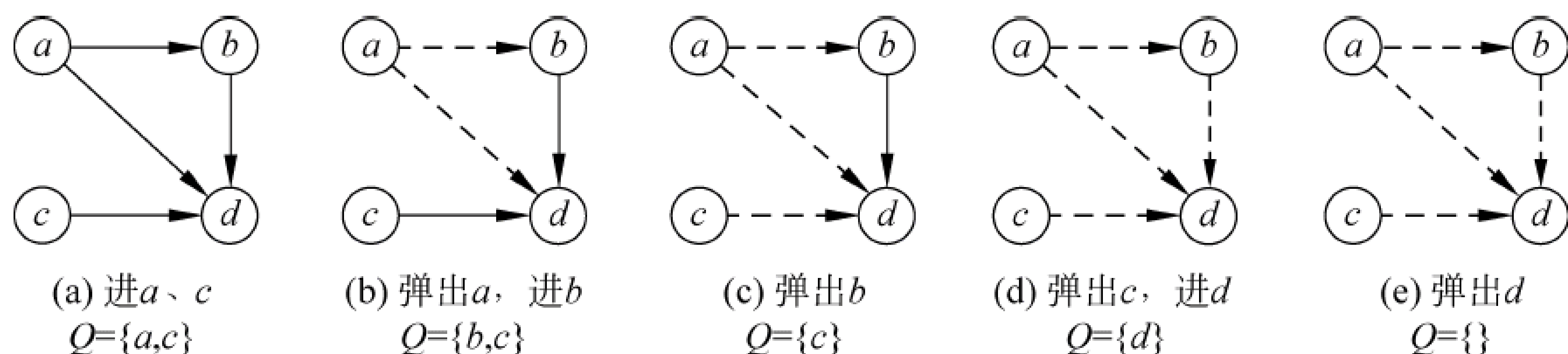


图10.12 输出字典序的拓扑排序

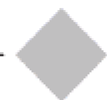
如果不用优先队列找最小的点，而是用暴力查找或者排序算法，效率会比较低，读者可以试一试。

用DFS可以输出字典序吗？思考上述解题的过程可以发现，用DFS是不行的。上面处理的过程相当于把点按优先级分成不同的层次，在每个层次都要把这一层入度减为0的点按大小顺序输出；而DFS是深度搜索，处理的是上下层之间的关系，不能处理这种同层次的关系。读者可以自己画一个比较复杂的多层次的图加深理解。

【习题】

poj 1270“Following Orders”，按字典序从小到大输出所有拓扑排序。这一题很重要。

hdu 3342“Legal or Not”，hdu 2647“Reward”、hdu 5695“Gym Class”，简单拓扑排序。



hdu 4857“逃生”,反向建图。

hdu 1811“Rank of Tetris”,并查集＋拓扑排序。

10.5　欧　拉　路

欧拉路是简单的图问题,和拓扑排序一样,也用DFS直接实现。

读者小时候可能玩过“一笔画”游戏:给一个图,要求一笔连续地画出整个图,必须经过每条边,并且只能经过一次,点可以重复经过。

这个问题来自于中世纪数学家欧拉的七桥问题。这条一笔画路线称为欧拉路。如果还要求起点和终点相同,则称为欧拉回路。

欧拉路:从图中某个点出发遍历整个图,图中的每条边通过且只通过一次。

欧拉回路:起点和终点相同的欧拉路。

欧拉路问题主要有两个,即是否存在欧拉路和打印出欧拉路。问题的解决主要通过处理度(degree)。一个点上连接的边的数量称为这个点的度数。在无向图中,如果度数是奇数,这个点称为奇点,否则称为偶点。在有向图中有出度和入度。

1. 欧拉路和欧拉回路是否存在

首先,图应该是连通图。在编程时用DFS或者并查集来判断连通性。

其次,判断图是否存在欧拉路或欧拉回路:

(1) 无向连通图的判断条件。如果图中的点全都是偶点,则存在欧拉回路;任意一点都可以作为起点和终点。如果只有两个奇点,则存在欧拉路,其中一个奇点是起点,另一个是终点。不可能出现有奇数个奇点的无向图,请读者思考。

(2) 有向连通图的判断条件。把一个点上的出度记为1、入度记为－1,这个点上所有的出度和入度相加就是它的度数。一个有向图存在欧拉回路,当且仅当该图所有点的度数为0。如果只有一个度数为1的点、一个度数为－1的点,其他所有点的度数为0,那么存在欧拉路径,其中度数为1的是起点、度数为－1的是终点。

下面用一个简单题讲解欧拉路的判断。

uva 10054“The Necklace”

有n个珠子。每个珠子有两种颜色,分布在珠子的两边。一共有50种不同的颜色。把这些珠子串起来,要求两个相邻的珠子接触的那部分颜色相同。问是否能连成一个珠串项链?如果能,打印一种连法。

这一题是典型的无向图求欧拉回路。

首先,判断所有的点是否为偶点,如果存在奇点,则没有欧拉回路;其次,判断所给的图是否连通,不连通也不是欧拉回路。

下面的程序只判断了有无欧拉回路。关于连通性,读者可以自己用DFS或者并查集实现(此题很简单,没用到DFS和并查集)。

这一题需要注意的是可能有重边,即邻居点u、v之间可能有多个边。

uva 10054 题：判断欧拉回路

```
for(i = 1; i <= n; i++) {              //输入图,用邻接矩阵 G[][]存图
    scanf("%d%d", &u, &v);
    degree[u]++;
    degree[v]++;                       //记录点的度
    G[u][v]++;
    G[v][u]++;                         //0: 不连接; 1: 连接; >1: 有重边
}
for(i = 1; i <= n; i++)
    if(d[i] % 2)  break;               //存在奇点,退出; 无欧拉回路
```

2. 输出一个欧拉回路

对一个无向连通图做 DFS 就输出了一个欧拉回路。

uva 10054 题：输出一个欧拉回路

```
void euler(int u){                     //从 u 开始 DFS
    int v;
    for(v = 1; v <= 50; v++)           //深搜 u 的所有邻居
      if(G[u][v]) {
        G[u][v]--; G[v][u]--;          //可能有重边
        euler(v);
        printf("%d %d\n", v, u);       //请思考为什么在 euler(v)后面打印
      }
}
```

图 10.13 可以帮助读者理解上面的程序。

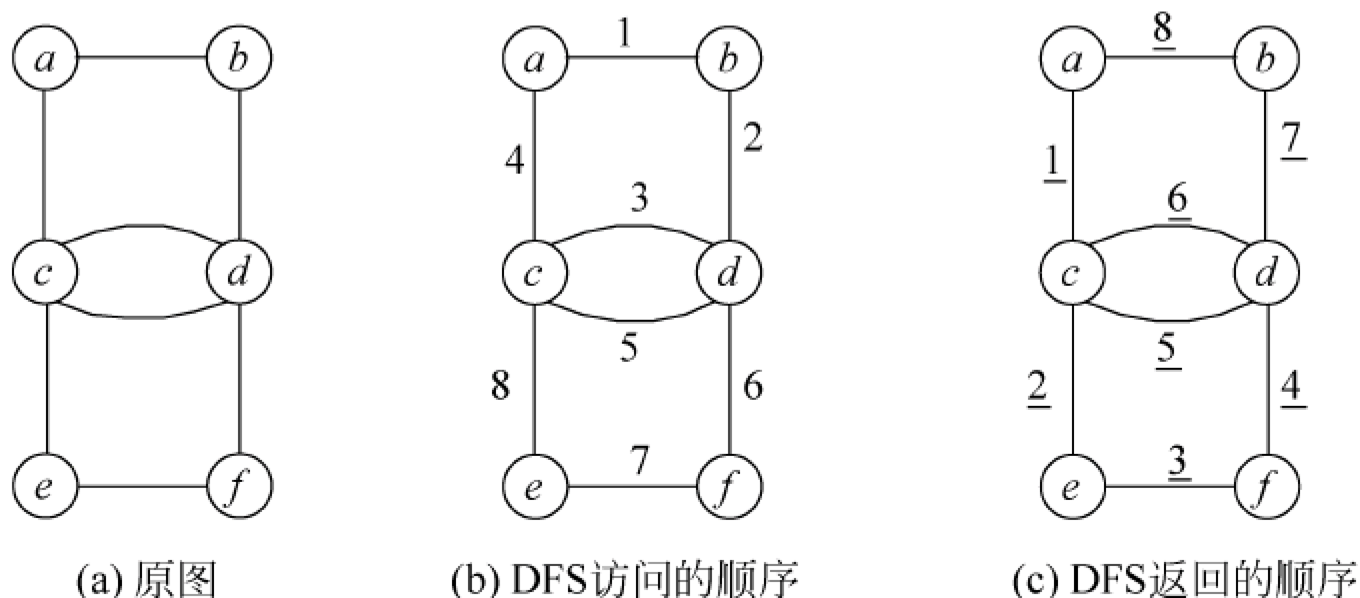


图 10.13 输出一个欧拉回路

从图 10.13(a)中的 a 点开始 DFS,DFS 的对象是边。图 10.13(b)边上的数字是 DFS 访问的顺序,也可以有别的顺序,图中为了帮助读者理解,特意选了一个不太“好”的顺序。图 10.13(c)边上画线的数字是 DFS 返回的顺序,它正好是一个欧拉回路。

程序中输出的路径实际上是从终点到起点的一条路径,对于无向图来说,因为起点和终点都是一个点,所以并没有关系。

如果是有向图,那么输出的是一个逆序的路径,可以用栈把逆序按正序打印出来,参考“拓扑排序”中打印路径时对栈的使用。

在上面的程序中，图是用邻接矩阵表示的。作为练习，读者可以用邻接表重写程序。

3. 用非递归 DFS 输出欧拉回路

上面用递归实现的 DFS 输出欧拉回路。递归常见的问题是爆栈，如果数据很大，就不能直接用递归，需要自己写个栈模拟递归。请读者练习下面的题目。



视频讲解

（1）poj 1780“code”。输入数字位数 n，输出一串数字，其中包含所有可能的 n 位数字序列，而且只包含一次；用字典序输出。

分析：欧拉回路问题。但是 $n=10^6$，会爆栈。

（2）hdu 4850“Wow！Such String！”。用 26 个小写字母构造一个长度为 n 的串，其中任意长度为 4 的子串都不相同；$n \leqslant 500\ 000$。

分析：本题可能有 4^{26} 个子串。这一题有不同解法，如果用 DFS，也容易爆栈。

4. 混合图欧拉路问题

有的图不是单纯的有向图或无向图，而是二者的混合，同时存在有向边和无向边。这是一个比较困难的问题，需要用最大流求解，具体内容见“10.11.3 Dinic 算法和 ISAP 算法”。

【习题】

hdu 1878“欧拉回路”。判断是否存在回路，无向图。

hdu 1116“Play on Words”。首尾连单词，有向图，可以分别用 DFS 和并查集判断连通性。

hdu 5883“The Best Path”。无向图欧拉路。

10.6　无向图的连通性

10.6.1　割点和割边

在无向图中，所有能互通的点组成了一个“连通分量”。在一个连通分量中有一些关键的点，如果删除它，会把这个连通分量分成两个或更多，这种点称为割点（Cut vertex）。

类似的有割边（Cut edge，又称为桥，bridge）问题。在一个连通分量中，如果删除一个边，把这个连通分量分成了两个，这个边称为割边。

研究割点和割边是很有意义的。从割点、割边扩展出**双连通**问题，即如何实现一个没有割点和割边的图。例如在计算机网络中，可靠性是重要的问题，希望能在某些网络结点出故障的情况下不影响整个网络的通畅。那么，应该如何布置网络才能不出现割点，并且部署的结点最少？

本节先研究一个基本问题：在一个无向连通图 G 中有多少个割点？

暴力方法：删除每个点，然后用 DFS 求连通性，如果连通分量变多，那么就是割点。其复杂度是 $O(V(V+E))$，不是好算法。

下面介绍用 DFS 求割点的算法，即利用“深搜优先生成树”求割点。请读者先回顾 10.3 节的概念。

在一个连通分量 G 中，对任意一个点 s 做 DFS，能访问到所有点，产生一棵“深搜优先生成树” T。那么对 G 求割点，和 T 有什么关系呢？

定理 10.1：T 的根结点 s 是割点，当且仅当 s 有两个或更多的子结点。这个定理很容易理解，如果 s 是割点，它会把图分成不相连的几部分，这几个部分都会生成子树；如果 s 不是割点，它只会连接一个子树。

读者可以验证图 10.14。图(b)是 a 的生成树，a 点是割点，它有子结点 b 和 c。图中点上面的数字是递归的顺序，下面画线的数字是递归返回的顺序。

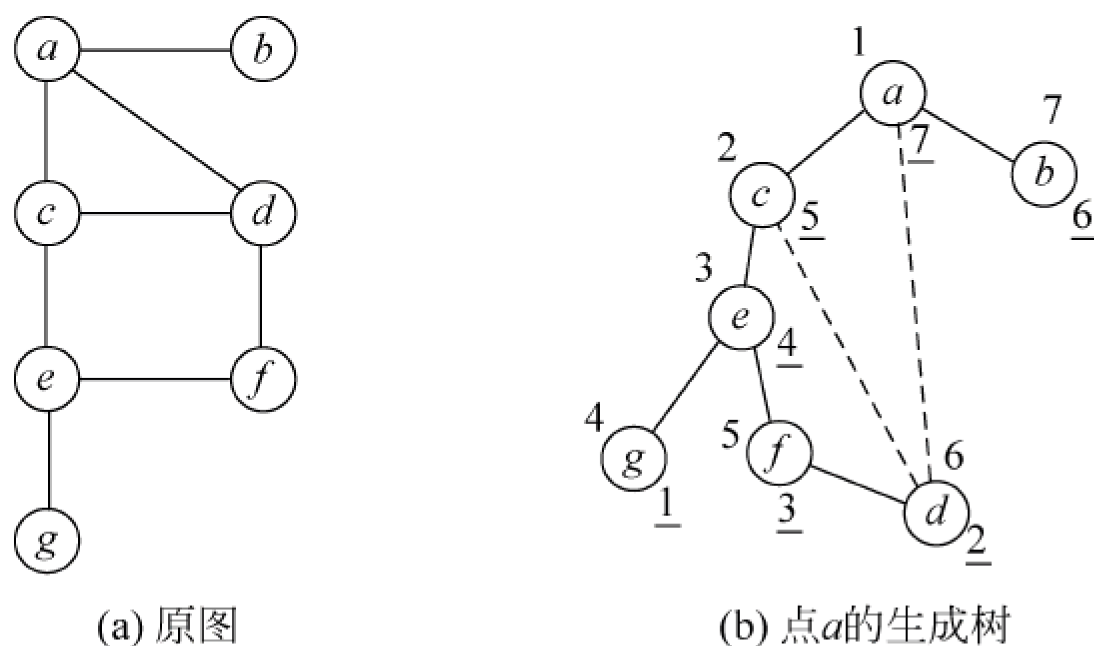


图 10.14　根结点是割点的判断

b 不是割点，如果用 b 生成树，只有一个子结点 a。

定理 10.2：T 的非根结点 u 是割点，当且仅当 u 存在一个子结点 v，v 及其后代都没有回退边连回 u 的祖先。这个定理也容易理解，如果 u 是割点，它会把图分成两部分或更多，其中至少一个后代肯定没有通过其他边(回退边，即绕过 u 回去的边)连回 u 的祖先，否则图就不会被分开了。

例如图 10.14(b)中的 c 点，它的子结点只有一个 e，而 e 后面有个子结点 d 有回退边连回了根结点 a，所以 c 不是割点。再看 e 点，有一个子结点 g，没有回退边连回 e 的祖先，所以 e 是割点。

如何编程实现定理 10.2?

设 u 的一个直接后代是 v。

定义 num[u]，记录 DFS 对每个点的访问顺序，num 值随着递推深度增加而变大。

定义 low[v]，记录 v 和 v 的后代能连回到的祖先的 num。

只要 low[v]$\geqslant$num[u]，就说明在 v 这个支路上没有回退边连回 u 的祖先，最多退到 u 本身。这就是定理 10.2。

下面的图 10.15 是例子，low[u]的初始值等于 num[u]，即连到自己。图 10.15(a)没有回退边。b 的后代是 c，low[c]=3，num[b]=2，有 low[c]$\geqslant$num[b]，说明 b 的支路 c 上没有回退边连回去，所以 b 是割点。

在图 10.15(b)中，观察 low[]是如何更新的。最后访问的 d 是递归最深处的点，它的 num[d]=4，它有回退边连到 b，low[d]的初始值是 4，更新为 low[d]=num[b]=2，表示有回退边到 b。然后 d 递归回到 c，low[c]更新为 low[c]=low[d]=2，表示 c 通过后代能回退到 b。以上是 low[]的更新过程。继续考察 c：由于 low[d]=2，num[c]=3，说明 c 的后代 d 有回退边连到了 c 的祖先，所以 c 不是割点。

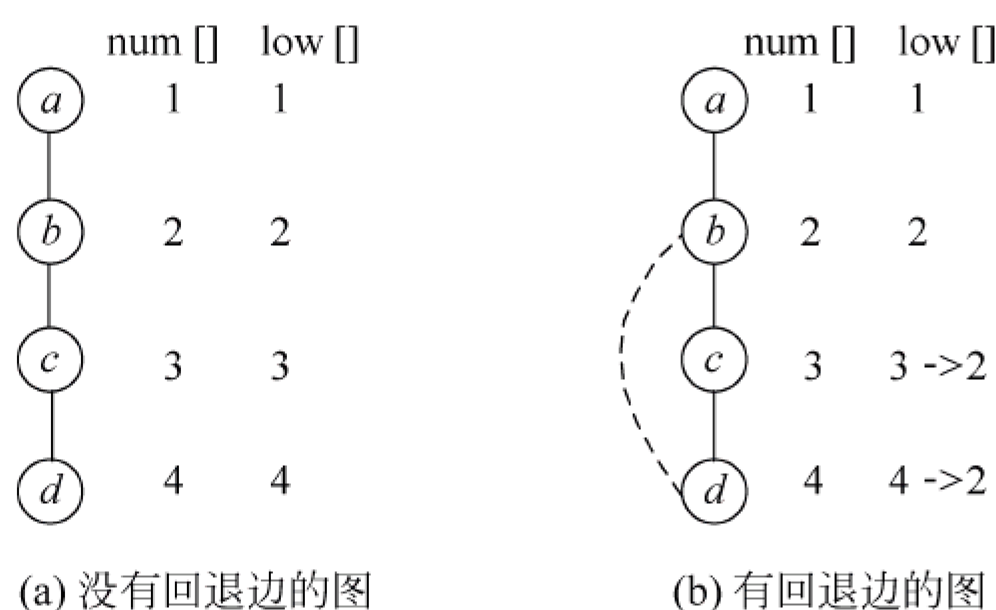


图 10.15 非根结点是割点的判断

特别有意思的是，上述判断割点的条件 low[v]≥num[u]只要改为 low[v]>num[u]就能用于**判断割边**。这表示 u 的支路 v 以及 v 的后代只能回退到 v，而到不了 u，那么边(u,v)肯定就是割边。例如图 10.15(b)中的 b 点，有 low[c]=2，num[b]=2，说明(b,c)不是割边；再看 a 点，有 low[b]=2，num[a]=1，low[b]>num[a]，所以(a,b)是割边。

poj 1144 "network"

输入一个无向图，求割点的数量。

一个电话线公司正在建立一个电话线缆网络。他们用线缆连接了若干个地点，这些线是双向的，每个地点都有一个电话交换机。从每个地点都能通过线缆到达其他任意的地点，并不一定直接连接，可以通过若干个交换机来到达目的地。有时候某个地点供电出问题，交换机会停止工作。工作人员意识到，除非这个地点是不可达的，否则它还会导致一些其他的地点不能互相通信。称这个地点为关键点。工作人员想要写一个程序找到所有关键点的数量。

在下面的程序中，用 int dfn 记录进入递归的顺序(也称为时间戳)，然后赋值给这个递归中点 u 的 num[u]。

poj 1144 部分代码

```
const int N = 109;
int low[N], num[N], dfn;                   //dfn 记录递归的顺序,用于给 num 赋值
bool iscut[N];
vector < int > G[N];                       //存图
void dfs(int u, int fa){                   //u 的父结点是 fa
low[u] = num[u] = ++ dfn;                  //初始值
int child = 0;                             //孩子数目
for (int i = 0;i < G[u].size(); i++){      //处理 u 的所有子结点
    int v = G[u][i];
    if (!num[v]) {                         //v 没访问过
        child++;
        dfs(v, u);
        low[u] = min(low[v], low[u]);      //用后代的返回值更新 low 值
        if (low[v] >= num[u] && u != 1)
            iscut[u] = true;               //标记割点
```

```
    }
    else if(num[v] < num[u] && v != fa)
              //处理回退边,注意这里 v != fa,fa 是 u 的父结点
              //fa 也是 u 的邻居,但是前面已经访问过,不需要处理它
        low[u] = min(low[u], num[v]);
}
if (u == 1 && child >= 2)                    //根结点,有两个以上不相连的子树
    iscut[1] = true;
}
int main(){
    int ans, n;
    //在这里输入图,程序略
    memset(low, 0, sizeof(low));
    memset(num, 0, sizeof(num));
    dfn = 0;
    memset(iscut, false, sizeof(iscut));
    ans = 0;
    dfs(1, -1);                              //DFS 的起点是 1
    for(int i = 1;i <= n;i++)  ans += iscut[i];
    printf("%d\n", ans);
}
```

判断割边。把程序中的 if (low[v] >= num[u] && u != 1)改为 if (low[v]>num[u] && u != 1),其他程序不变,就是求割边的数量。

10.6.2 双连通分量

在一个连通图中选任意两点,如果它们之间至少存在两条"点不重复"的路径,称为点双连通。一个图中的点双连通极大子图称为"点双连通分量"(block,或者 2-connected component)。点双连通分量是一个"可靠"的图,去掉任意一个点,其他点仍然是连通的。也就是说,点双连通分量中没有割点。

类似地有"边双连通分量",如果任意两点之间至少存在两条"边不重复"的路径,称为"边双连通"。在边双连通图中去掉任意一个边,图仍然是连通的。也就是说,边双连通图中没有割边。

1. 点双连通分量

在一个无向图 G 中有多少个点双连通分量?

求解点双连通分量和求割点密切相关。不同的点双连通分量最多只有一个公共点,即某一个割点;任意一个割点都是至少两个点双连通分量的公共点。

计算点双连通分量一般用 Tarjan 算法[①],下面是算法的思路。

前面讲解了如何用 DFS 进行割点的计算,可以发现,在找到一个割点的时候已经完成了一次对某个极大点双连通子图的访问。那么,在进行 DFS 的过程中,把遍历过的点保存起来,就可以得到这个点双连通分量。

① Tarjan 提出了很多算法,这是其中之一。

DFS 的访问过程用栈来保存是最合理的，所以，在求解割点的过程中，用一个栈保存遍历过的边，然后每当找到一个割点，即满足关系 low[v] >= num[u]的点 u，就将栈里的边拿出来。

注意，放入栈中的不是点，而是边。因为一个边只属于一个点双连通分量，而一个割点属于多个点双连通分量，如果进入栈中的是点，这个割点弹出来之后就只能给一个点双连通分量了，它连接的其他点双连通分量就会少了这个点。

练习题：poj 1523 "SPF"，一个图中有多少个割点？每个割点能把网络分成几个点双连通分量？

2. 边双连通分量

给定一个图 G，它有多少个边双连通分量？至少应该添加多少条边，才能使任意两个边双连通分量之间都是双连通的，也就是使图 G 是双连通的？

> **poj 3352 "Road Construction"**
>
> 给定一个无向图 G，图中没有重边。问添加几条边才能使无向图变成边双连通图。

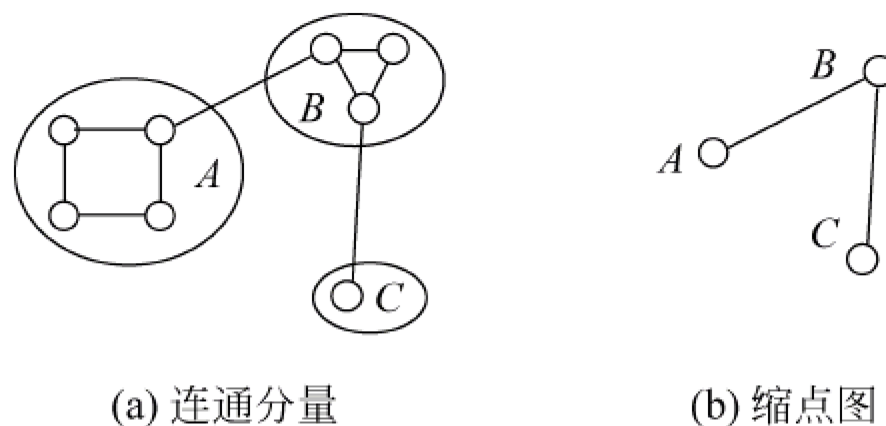
边双连通分量的计算用到了"缩点"的技术。

(1) 首先找出图 G 的所有边双连通分量。

在 DFS 过程中，图 G 所有的点都生成一个 low 值，low 值相同的点必定在同一个边双连通分量中。DFS 结束后，有多少 low 值就有多少个边双连通分量。

(2) 把每一个边双连通分量都看作一个点，即把那些 low 值相同的点合并为一个"缩点"。这些缩点形成了一棵树，例如图 10.16。

(3) 问题被转化为：至少在缩点树上增加多少条边才能使这棵树变为一个边双连通图。容易推导出：至少增加的边数＝(总度数为 1 的结点数＋1)/2。例如图 10.16(b)有两个度数为 1 的点 A、C，至少增加的边数＝(2＋1)/2＝1。



(a) 连通分量　　(b) 缩点图

图 10.16　边双连通分量的缩点

poj 3352 程序

```
#include <cstring>
#include <vector>
#include <stdio.h>
using namespace std;
const int N = 1005;
int n, m, low[N], dfn;
vector<int> G[N];                          //存图
void dfs(int u,int fa){                    //计算每个点的 low 值
   low[u] = ++dfn;
   for(int i=0;i<G[u].size();i++){
       int v = G[u][i];
       if(v == fa) continue;
       if(!low[v])
```

```
            dfs(v,u);
        low[u] = min(low[u], low[v]);
    }
}
int tarjan(){
    int degree[N];                              //计算每个缩点的度数
    memset(degree,0,sizeof(degree));
    for(int i=1; i<=n; i++)                     //把有相同 low 值的点看成一个缩点
        for(int j=0; j<G[i].size(); j++)
            if(low[i] != low[G[i][j]])
                degree[low[i]]++;
    int res=0;
    for(int i=1;i<=n;i++)                       //统计度数为 1 的缩点的个数
        if(degree[i]==1) res++;
    return res;
}
int main(){
    while(~scanf("%d%d", &n, &m)){
        memset(low, 0, sizeof(low));
        for(int i=0; i<=n; i++)  G[i].clear();
        for(int i=1; i<=m; i++){
            int a, b;
            scanf("%d%d", &a, &b);
            G[a].push_back(b);  G[b].push_back(a);
        }
        dfn = 0;
        dfs(1,-1);
        int ans = tarjan();
        printf("%d\n",(ans+1)/2);
    }
    return 0;
}
```

【习题】

hdu 3394 “Railway”，点双连通分量。

hdu 3749 “Financial Crisis”，点双连通分量。

hdu 2460 “Network”，边双连通分量。

hdu 4587 “TWO NODES”，无向图求割点。

10.7 有向图的连通性

本节的内容与拓扑排序的思想有关，读者在阅读之前请先认真学习本章“10.4 拓扑排序”的内容。

强连通。在有向图 G 中，如果两个点 u、v 是互相可达的，即从 u 出发可以到达 v，从 v

出发也能到达 u，则称 u 和 v 是强连通的。如果 G 中的任意两个点都是互相可达的，称 G 是强连通图。

强连通分量。如果一个有向图 G 不是强连通图，那么可以把它分成多个子图，其中每个子图的内部是强连通的，而且这些子图已经扩展到最大，不能与子图外的任意点强连通，称这样的一个"极大强连通"子图是 G 的一个强连通分量(Strongly Connected Component，SCC)。

一个常见的问题：G 中有多少个 SCC？在解决这个问题之前需要研究 SCC 的特征。

(1) 出度和入度。一个点必须有出发的边，也有到达的边，这样才会与其他点强连通。

(2) 把一个 SCC 从 G 中挖掉，不影响其他点的强连通性。可以把图上的一个个 SCC 想象成一个个岛，岛内部是强连通的；岛之间只有单向道路连接，不会形成环路。把每个岛虚拟成一个点，那么所有这些虚拟点构成的虚拟图是一个有向无环图 DAG；这个虚拟 DAG 图中的点与其他点都不是强连通的，DAG 中的虚拟点的数量就是 SCC 的数量，如图 10.17 所示。可以推论出，每个岛都可以挖掉，而不会影响其他岛内部的连通性。

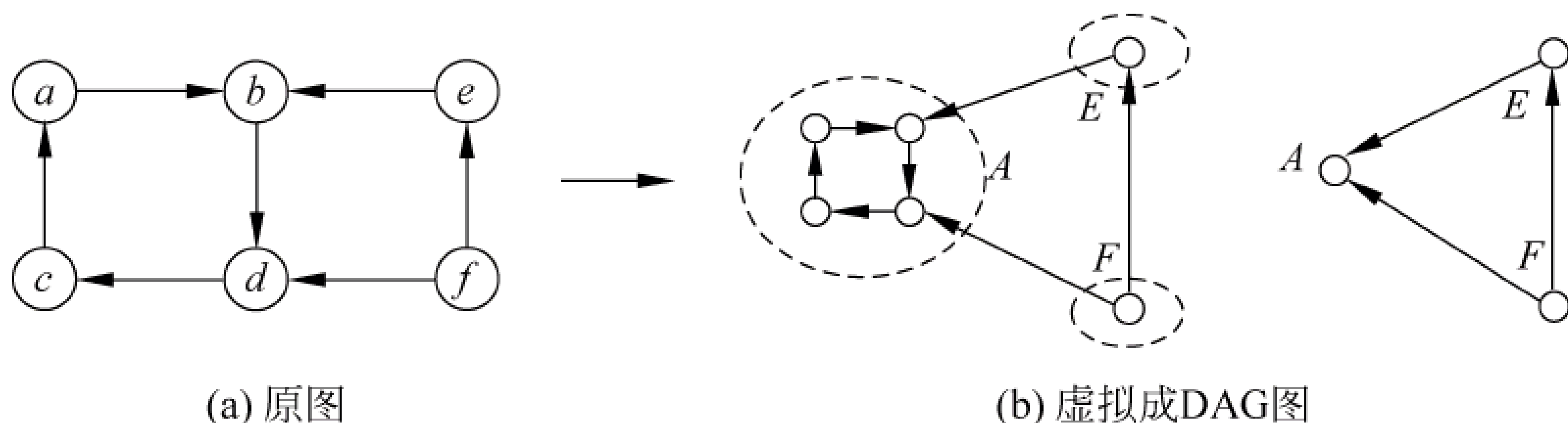


图 10.17　SCC 的虚拟图

用暴力的方法求 SCC 是对每个点求连通性，然后进行比较，那些互相连通的点就组成了 SCC。这可以通过对每个点都进行 DFS 或者 BFS 搜索得到，例如对图 10.17(a)进行搜索的结果如下：

分别从 a、b、c、d 点出发，可以到达：$\{a,b,c,d\}$；

从 e 点出发可以到达：$\{a,b,c,d,e\}$；

从 f 点出发可以到达：$\{a,b,c,d,e,f\}$。

最少的 $\{a,b,c,d\}$ 是一个强连通分量，从整个图中挖掉它，剩下最小的是 $\{e\}$，再挖掉它，最后是 $\{f\}$，得到 3 个 SCC，即 $\{a,b,c,d\}$、$\{e\}$、$\{f\}$。

暴力法的复杂度是 $O(V^2+E)$。

求 SCC 有 3 种高效算法，即 Kosaraju、Tarjan、Garbow，它们的复杂度都是 $O(V+E)$，但 Kosaraju 要差一些。下面介绍 Kosaraju、Tarjan 算法。

10.7.1　Kosaraju 算法

Kosaraju 算法用到了"反图"的技术，基于下面两个原理：

(1) 一个有向图 G，把 G 所有的边反向，建立反图 rG，反图 rG 不会改变原图 G 的强连通性。也就是说，图 G 的 SCC 数量与 rG 的 SCC 数量相同。这里直接用上面的虚拟 DAG 图做例子，图 10.18(a)中的 A、E、F 是 3 个 SCC，内部的点都是强连通的。

(2) 对原图 G 和反图 rG 各做一次 DFS，可以确定 SCC 数量。

对原图 G 做 DFS 是为了确定点的先后顺序。可以发现，对生成的虚拟 DAG 图，可以用 DFS 做拓扑排序，排序结果是 F、E、A（不过，此时并没有确定哪些点是属于 A、E、F 的）。而且，F 内部优先级最高的那个点，优先级高于 E、A 内部所有的点；E 内部优先级最高的那个点，优先级高于 A 内部所有的点。这个有用的结果将用于下面的步骤。

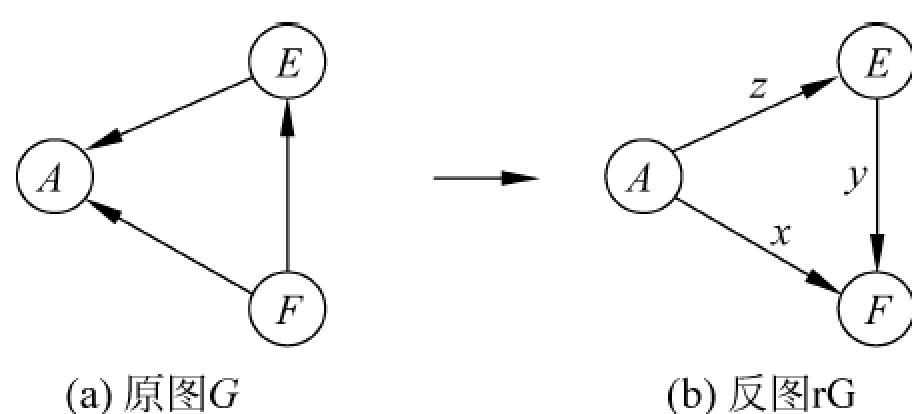


图 10.18 原图与反图

确定了顺序，然后从优先级最高的点（这个点属于 F）开始，在反图上做 DFS。为什么要在反图上做 DFS？这样做可以求得被隔离的“岛”。例如求 F 包含哪些点，想办法把 F 和其他点隔离就好了；原图中 F 是只有出度的点，改成反图后，F 变成了只有入度的点，那么从 F 出发做 DFS，就会被反边 **x、y 堵住**，DFS 搜索到的点被限制在 F 内。显然，只能搜到并且能全部搜到 F 内部的点，而无法到达 A、E，这样就确定了 F，也就是确定了第 1 个 SCC。

下一步，删除 F，然后继续在剩下的优先级最高的点开始搜，这一步搜到的点属于 E，而 E 也被反边 **z 堵住**，只能搜到属于 E 的点，确定了第 2 个 SCC。最后，删除 E，确定属于 A 的点，也就是确定了第 3 个 SCC。

算法步骤如下：

（1）在 G 上做一次 DFS，标记点的先后顺序。在 DFS 的过程中标记所有经过的点，把递归到最底层的那个点标记为最小，然后在回退的过程中，其他点的标记逐个递增。和上节拓扑排序中的 DFS 操作一样，并不需要找一个特殊的点作为起点，可以想象有一个起点 v，v 连接所有的结点，从 v 开始 DFS。

在图 10.19(a)中，从虚拟的点 v 出发，按 a、b、c、d、e、f 的顺序执行 DFS，DFS 返回的结果是 c、d、b、a、e、f；每个点的大小标记见图中的数字。如果搜索顺序不同，结果也会不同；但是，不管是什么顺序，**f 的标记肯定最大**，这是拓扑排序的原理。读者可以试试其他顺序，验证这个结论。



视频讲解

（2）在反图 rG 上再做一次 DFS，顺序从标记最大的点开始到最小的点。首先是点 f，记录所有它能到达的点，这些点组成了第 1 个 SCC，图 10.19(b)中点 f 只能到达自己，这是第 1 个 SCC；然后删除 f，从剩下的最大的点继续 DFS，这次是点 e，是第 2 个 SCC；最后从点 a 开始搜，返回$\{c,d,b,a\}$，这是第 3 个 SCC。

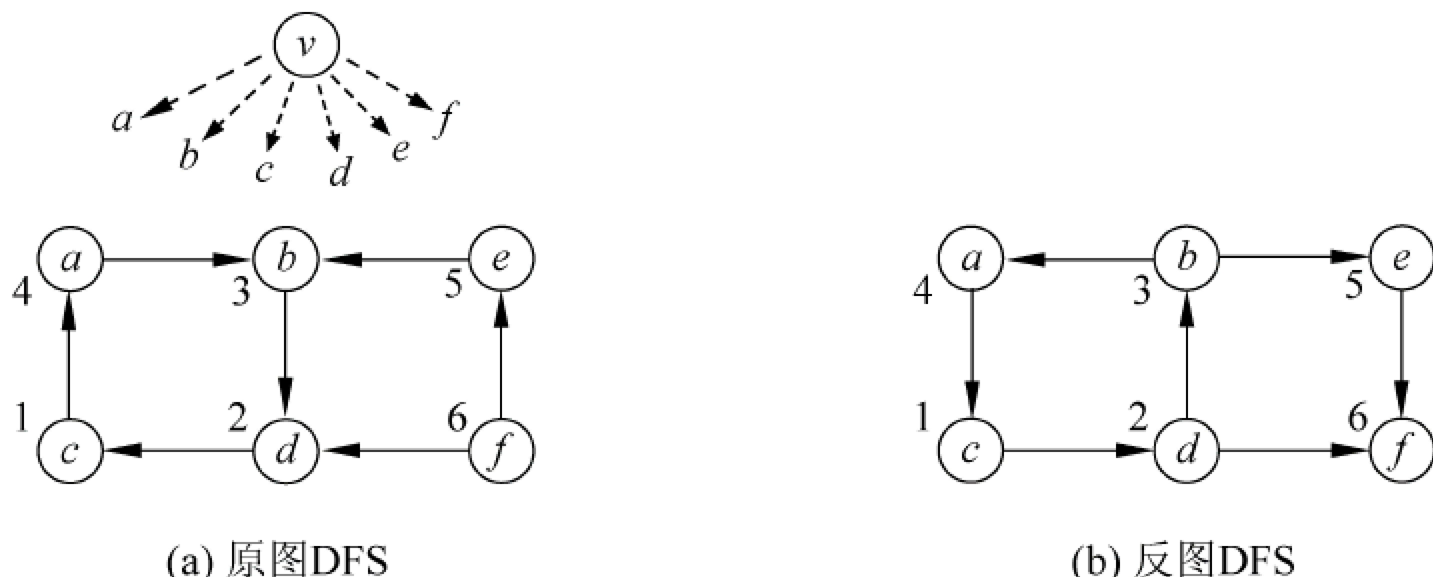
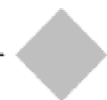


图 10.19 原图和反图的 DFS



hdu 1269 "迷宫城堡"

一个有向图，有 n 个点（$n \leqslant 10\,000$）和 m 条边（$m \leqslant 100\,000$）。判断整个图是否强连通，如果是，输出 Yes，否则输出 No。

hdu 1269 的 Kosaraju 算法代码①

```
include <bits/stdc++.h>
using namespace std;
const int NUM = 10005;
vector<int> G[NUM], rG[NUM];
vector<int> S;                              //存第一次 dfs1()的结果: 标记点的先后顺序
int vis[NUM], sccno[NUM], cnt;              //cnt: 强连通分量的个数
void dfs1(int u) {
    if(vis[u]) return;
    vis[u] = 1;
    for(int i=0; i<G[u].size(); i++)   dfs1(G[u][i]);
    S.push_back(u);                         //记录点的先后顺序,标记大的放在 S 的后面
}
void dfs2(int u) {
    if(sccno[u]) return;
    sccno[u] = cnt;
    for(int i=0; i<rG[u].size(); i++)   dfs2(rG[u][i]);
}
void Kosaraju(int n) {
    cnt = 0;
    S.clear();
    memset(sccno, 0, sizeof(sccno));
    memset(vis, 0, sizeof(vis));
    for(int i = 1; i<=n; i++)   dfs1(i);   //点的编号: 1~n.递归所有点
    for(int i = n-1; i>=0; i--)
        if(!sccno[S[i]]) {cnt++; dfs2(S[i]);}
}
int main(){
    int n, m, u, v;
    while(scanf("%d%d", &n, &m), n != 0 || m != 0) {
        for(int i = 0; i<n; i++) {G[i].clear(); rG[i].clear();}
        for(int i = 0; i<m; i++){
            scanf("%d%d", &u, &v);
            G[u].push_back(v);              //原图
            rG[v].push_back(u);             //反图
        }
        Kosaraju(n);
        printf("%s\n", cnt == 1 ? "Yes" : "No");
    }
    return 0;
}
```

① 部分代码参考《算法竞赛入门经典训练指南》，刘汝佳，清华大学出版社，320 页。

该程序用 cnt 记录 SCC 的数量，并且统计了每个点所属的 SCC，sccno[i]是第 i 个点所属的 SCC。在 dfs2()中，sccno[i]也被用于记录点 i 是否被访问，如果 sccno[i]不等于 0，说明它已经被处理过；在 dfs1()中，用 vis[i]记录点 i 是否被访问。

Kosaraju 算法的复杂度是 $O(V+E)$。

10.7.2 Tarjan 算法

上面的 Kosaraju 算法，其做法是从图中一个一个地把 SCC"挖"出来。Tarjan 算法能在一次 DFS 中把所有点都按 SCC 分开。这并不是不可思议的，它利用了 SCC 的如下特点。

定理 10.3：一个 SCC，从其中任何一个点出发，都至少有一条路径能绕回到自己。

在继续讲解之前，请读者先回顾无向图 DFS 中求割点的 low[]和 num[]操作。Tarjan 算法用到了同样的技术，这个技术结合定理 10.3 就是 Tarjan 算法。

下面是例子，图 10.20 中有 3 个 SCC，即{a,b,d,c}、{e}、{f}。

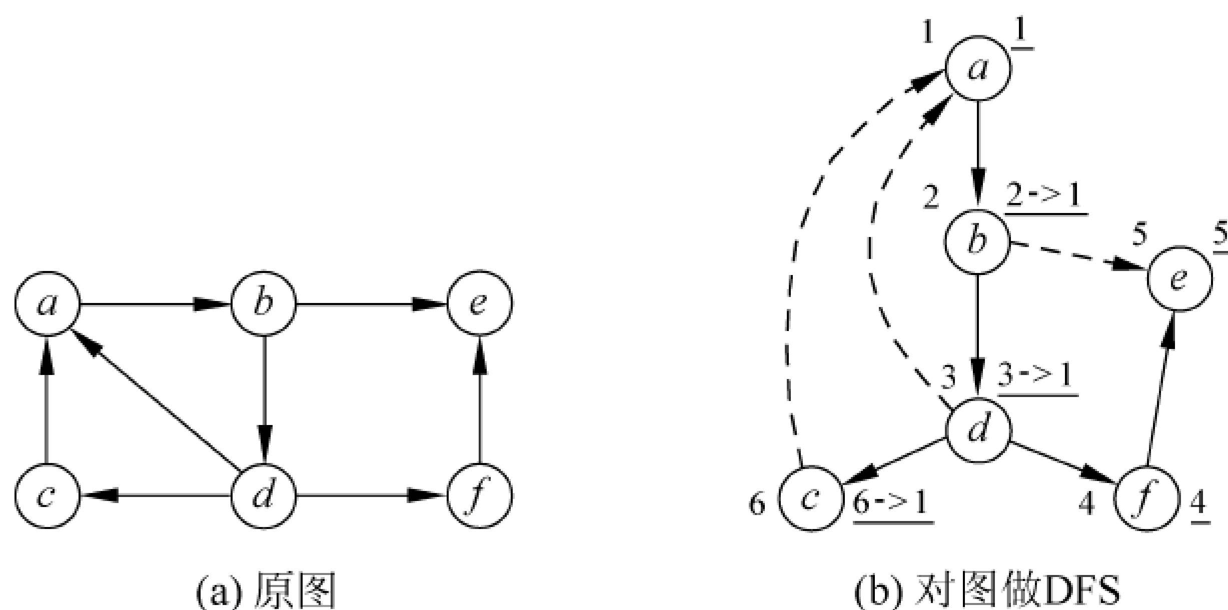


图 10.20 SCC 的 low[]和 num[]操作

图 10.20(a)是原图。图(b)对它做 DFS，每个点左边的数字标记了 DFS 访问它的顺序，即 num[]值，右边的画线数字是 low[]值，即能返回到的最远祖先。每个点的 low[]初始值等于 num[]，即连到自己。观察 c 的 low[]值是如何更新的：它的初始值是 6，然后有一个回退边到 a，所以更新为 1；它的递归祖先 d、b 的 low[]值也跟着更新为 1。e 和 f 的 low[]值不能更新。

图 10.20(b)是从 a 开始 DFS 的，a 成为{a,b,d,c}这个 SCC 的共同祖先。其实，从{a,b,d,c}中任意一个点开始 DFS，这个点都会成为这个 SCC 的祖先。认识到这些，可以帮助读者理解后面的解释：可以用栈分离不同的 SCC。

图 10.20(b)中的 low[]值有 3 个部分，即等于 1 的{a,b,d,c}、等于 4 的{f}、等于 5 的{e}。这就是 3 个 SCC。

完成以上步骤，似乎已经解决了问题。每个点都有了自己的 low[]值，相同 low[]值的点属于一个 SCC。那么只要再对所有点做一个查询，按 low[]值分开就行了，其复杂度是 $O(V)$。其实有更好的办法，即在 DFS 的同时把点按 SCC(有相同的 low[]值)分开。

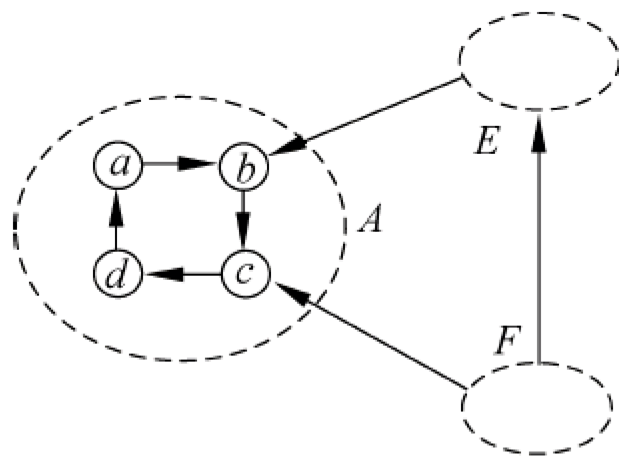


图 10.21 把图分成多个 SCC

以图 10.21 为例，其中有 3 个 SCC，即 A、E、F。假设从 F 中的一个点开始 DFS，DFS 过程可能会中途跳出 F，转入 A

或者 E，总之，最后会进入一个 SCC。

（1）假设 DFS 过程是 $F \to E \to A$，最后进入 A。

（2）在 A 这个 SCC 中将完成 A 内所有点的 DFS 过程，也就是说，最后的几步 DFS 会集中在 A 中的点 a、b、c、d。这几个点会计算得到相同的 low[]值，标记为一个 SCC，这样就好了。

（3）DFS 递归从 A 回到 E，并在 E 中完成 E 内部点的 DFS 过程。

（4）回到 F，在 F 内完成递归过程。

以上过程如何编程？读者能想起来，DFS 搜索是用递归实现的，而递归和栈这种数据结构在本质上是一致的。所以，可以**用栈来帮助处理**：

（1）从 F 开始递归搜索，访问到的某些点进入栈；

（2）E 中的某些点进入栈；

（3）在 DFS 的最底层，A 的所有点将被访问到并进入栈，当前栈顶的几个元素就是 A 的点，标记为同一个 SCC，并弹出栈；

（4）DFS 回到 E，在 E 中完成所有点的搜索并且入栈，当前栈顶的几个元素就是 E 的点，标记为同一个 SCC，并弹出栈；

（5）回到 F，完成 F 的所有点的搜索并且入栈，当前栈顶的几个元素就是 F 的点，标记为同一个 SCC，并弹出栈。结束。

为加深对上述过程中栈的理解，读者可以思考最先进入栈的点。每进入一个新的 SCC，访问并入栈的第一个点都是这个 SCC 的祖先，它的 num[]值等于 low[]值，这个 SCC 中所有点的 low[]值都等于它。

仍然以 hdu 1269 题为例，给出 Tarjan 算法代码。程序中用一个数组 int stack[N]模拟栈，读者可以尝试直接用 STL 的 stack<int>定义栈。

hdu 1269 的 Tarjan 算法代码

```
include <bits/stdc++.h>
using namespace std;
const int N = 10005;
int cnt;                                     //强连通分量的个数
int low[N], num[N], dfn;
int sccno[N], stack[N], top;                 //用 stack[ ]处理栈,top 是栈顶
vector <int> G[N];
void dfs(int u){
    stack[top++] = u;                        //u 进栈
    low[u] = num[u] = ++dfn;
    for(int i = 0; i < G[u].size(); ++i){
        int v = G[u][i];
        if(!num[v]){                         //未访问过的点,继续 DFS
            dfs(v);                          //DFS 的最底层,是最后一个 SCC
            low[u] = min(low[v], low[u]);
        }
        else if(!sccno[v])                   //处理回退边
            low[u] = min(low[u], num[v]);
    }
    if(low[u] == num[u]){                    //栈底的点是 SCC 的祖先,它的 low = num
```

```
            cnt++;
            while(1){
                int v = stack[--top];           //v弹出栈
                sccno[v] = cnt;
                if(u == v) break;               //栈底的点是SCC的祖先
            }
        }
    }
    void Tarjan(int n){
          cnt = top = dfn = 0;
          memset(sccno,0,sizeof(sccno));
          memset(num,0,sizeof(num));
          memset(low,0,sizeof(low));
          for(int i = 1; i <= n; i++)
              if(!num[i])
                dfs(i);
    }
    int main(){
        int n,m,u,v;
        while(scanf("%d%d", &n, &m), n != 0 || m != 0) {
            for(int i = 1; i <= n; i++){G[i].clear();}
            for(int i = 0; i < m; i++){
                scanf("%d%d", &u, &v);
                G[u].push_back(v);
            }
            Tarjan(n);
            printf("%s\n", cnt == 1 ? "Yes" : "No");
        }
        return 0;
    }
```

Tarjan 算法的复杂度也是 $O(V+E)$，但是它只做了一次 DFS，比 Kosaraju 算法快。

【习题】

hdu 1827 “Summer Holiday”，Tarjan 缩点。

hdu 3072 “Intelligence System”，Tarjan＋贪心。

hdu 3836 “Equivalent Sets”，给定有向图，至少要添加多少条边才能成为强连通图？

hdu 3639 “Hawk-and-Chicken”，强连通分量＋缩点。

hdu 3861 “The King’s Problem”，Tarjan＋最小路径覆盖。

hdu 1530 “Maximum Clique”，最大团简单题目。强连通分量的一个应用是最大团问题(Maximum Clique Problem，MCP)。

10.8　2-SAT 问题

2-SAT 问题可以用强连通分量和拓扑排序解决。

先用一个例子说明什么是 2-SAT 问题。

hdu 3062 “Party”

有 n 对夫妻被邀请参加一个聚会，每对夫妻中只有 1 人可以列席。在 $2n$ 个人中，某些人（不包括夫妻）之间有着很大的矛盾，有矛盾的两个人不会同时出现在聚会上。问有没有可能让 n 个人同时列席？

1. 数字逻辑的解法



视频讲解

如果学过计算机系的大二课程“数字逻辑”，可以用卡诺图帮助理解这个题目。

输入样例：有 3 对夫妻 A（包括 A 男和 A 女，B 和 C 也是）、B、C，其中 A 男和 B 女有矛盾，A 女和 C 女有矛盾，A 男和 C 男有矛盾。

输出：所有合法的出席情况。

分析如下：

(1) 夫妻不同时出席。例如，第 A 对夫妻，丈夫是 A，妻子是 $\bar{A}$，因为夫妻不同时出席，所以互为反变量[①]。

(2) 不同夫妻的限制条件。例如，A 男和 B 女（B 女用 $\bar{B}$ 表示）有矛盾，即 A 和 $\bar{B}$ 不会同时出现，有 $A\bar{B}=0$。一共有 3 个限制：$A\bar{B}=0$，$\bar{A}\bar{C}=0$，$AC=0$。用卡诺图表示，图 10.22(a) 中的 5 个 0 是 3 个限制填图的结果。

AB \ C	0	1
00	0	
01	0	
11		0
10	0	0

(a) 限制条件

AB \ C	0	1
00	0	1
01	0	1
11	1	0
10	0	0

(b) 完整卡诺图

图 10.22　用卡诺图求解 2-SAT 问题

图 10.22(b) 中等于 1 的方格就是可行的答案，一共有 3 个 1：$\bar{A}\bar{B}C$、$\bar{A}BC$、$AB\bar{C}$。也就是 3 个合法出席方案：A 女＋B 女＋C 男，A 女＋B 男＋C 男，A 男＋B 男＋C 女。

2-SAT 的可行解有多少个？在上面卡诺图的图解中可以发现，卡诺图的方格有 2^n 个，也就是说，可行解的数量是 $O(2^n)$ 的，复杂度很高，所以一般不会要求输出所有的解，只需要判断序列是否存在，或者只输出一个可行解。

2. 2-SAT 问题的定义

根据上面的例子，给出 SAT 问题的定义，它本身是一个数字逻辑问题：有 n 个布尔变量（布尔变量的特点是只有 0、1 两个值），其中一些布尔变量之间有限制关系；用所有 n 个布尔变量组成序列，使得其满足所有限制关系；判断序列是否存在。这就是 SAT（Satisfiability）问题。如果每个限制关系只涉及两个变量，则是 2-SAT 问题。

3. 用图论的方法解决 2-SAT 问题

(1) 首先，把矛盾关系用图来表示。

举一个简单例子。有两对夫妻 A、B，有两个限制：A、B 矛盾，A、$\bar{B}$ 矛盾。

先看 A、B 的矛盾，有两个推论：如果 A 确定出席，那么只能 $\bar{B}$ 出席，用 $A\rightarrow\bar{B}$ 表示，表

① 在数字逻辑中有 3 种基本逻辑操作，即与、或、非。非：$\bar{A}$ 是 A 的反变量。或：$A+\bar{A}=1$。与：$A\bar{A}=0$。

示"有 A 必有 $\overline{B}$";如果 B 确定出席,只能 $\overline{A}$ 出席,用 $B\to\overline{A}$ 表示。

A、B 这一对矛盾,推出了两个结果,这是因为 A、B 是对等的,所以产生的关系是**对称**的。见下面的有向图 10.23(a)。

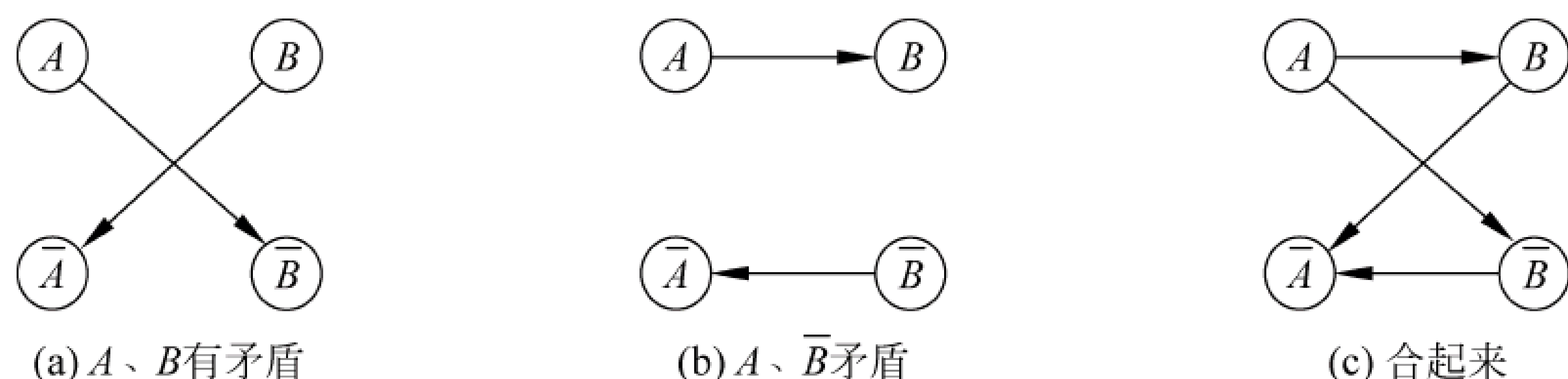


图 10.23　用图表示 A、B 的矛盾关系

同样,A、$\overline{B}$ 矛盾,推论是 $A\to B$、$\overline{B}\to\overline{A}$,见有向图 10.23(b)(可以观察到,这里推论出 A、B 同时出席,和前一个限制正好矛盾)。

两个限制合起来的有向图是图 10.23(c)。这个有向图的点包含了所有人,有向边说明了依赖关系。

(2) 合法的出席组合和强连通分量 SCC 的关系。

在最后的图 10.23(c)中,形成了多个强连通分量 SCC。一个 SCC 内部的点都是互相依赖的,也就是说,如果有一个出席,那么这个 SCC 内部的所有人都要出席。所以,一个 SCC 内部不应该有夫妻关系,因为夫妻只能出席一人。只要所有的 SCC 内部都没有夫妻,就会有合法的出席组合。为深入理解这一点,读者可以观察图 10.23(c),所有的点都不是强连通的,每个点都是独立的 SCC,所以这个图有合法的解。特别要注意其中有 $A\to B\to\overline{A}$,但 A 和 $\overline{A}$ 并不是强连通的。

所以,程序的步骤是根据给定的限制条件建图,计算 SCC,如果每个 SCC 内部都没有夫妻,就说明有合法的出席组合。

(3) 在图上求解一个合法组合。作为参照,读者可以先用上面卡诺图的方法得出有 $\overline{A}B$、$\overline{A}\overline{B}$ 两种出席组合。

读者可能觉得,只要在图 10.23(c)中沿着一条路径按顺序找,就能找到一个合法组合,因为一条路径上前后的点都是相互依赖的。但是,其实这个从前到后的顺序是不对的,应该按反序找,即从最后的点开始往前,这才是对的。这是因为最后的点是依赖性最大的,例如图 10.23(c)中的 $\overline{A}$,它被前面的 B 和 $\overline{B}$ 所依赖。

把每个 SCC 看成一个点,构成了一个 DAG 图,进行反图的拓扑排序,在选中点的时候同时排除图中相矛盾的点,就能找到合法的组合。

在编程时并不需要再做一次拓扑排序。在求 SCC 时已经得到了每个点所属的 SCC,SCC 的序号就是一个拓扑排序。

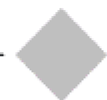
【习题】

hdu 3062 "Party",2-SAT 简单题。

hdu 1824 "Let's go home",简单题。

hdu 4115 "Eliminate the Conflict"。

hdu 4421 "Bit Magic"。



10.9 最 短 路

最短路径是图论中最为人们熟知的问题。

1. 最短路径问题

在一个图中有 n 个点、m 条边。边有权值,例如费用、长度等,权值可正可负。边可能是有向的,也可能是无向的。给定两个点,起点是 s,终点是 t,在所有能连接 s 和 t 的路径中寻找边的权值之和最小的路径,这就是最短路径问题。

2. 可加性参数和最小性参数

这两种参数区分了最短路径问题和网络流问题。

在最短路径问题中,是计算“路径上边的权值之和”。边的权值是“可加性参数”,例如费用、长度等,它们是“可加的”,一条路径上的总权值是这条路径上所有边的权值之和。下一节的“最小生成树”问题,边的权值也是“可加性参数”。

但是,在网络流问题中是找“路径上权值最小的边”。例如“最大流”问题,边的权值是“最小性参数”。比如水流,一条路径上的能流过的水流取决于这条路径上容量最小的那条边。再比如网络的带宽,一条网络路径上的整体带宽是这条路径上带宽最小的那条边的带宽。

3. 用 DFS 搜索所有的路径

在一般的图中,求图中任意两点间的最短路径,首先需要遍历所有可能经过的结点和边,不能有遗漏;其次,在所有可能的路径中查找最短的一条。如果用暴力法找所有路径,最简单的方法是把 n 个结点进行全排列,然后从中找到最短的。但是共有 $n!$ 个排列,是天文数字,无法求解。更好的办法是用 DFS 输出所有存在的路径,这显然比 $n!$ 要少得多,不过,其复杂度仍然是指数级的。

4. 用 BFS 求最短路径

在特殊的地图中,所有的边都是无权的,可以把每个边的权值都设成 1,那么 BFS 也是很好的最短路径算法,这些内容在 4.3.3 节中已经提到,请读者回顾有关内容。

下面讲解常见的 4 个最短路径算法。这几种方法差别很大,如果读者不能理解其思想,学起来容易头晕。为清晰地讲解这些算法,本书从 3 个方面展开:先结合现实中的模型讲解算法的思想;然后解释编程的逻辑过程;最后给出标准程序,这些程序结合了不同的数据结构和 STL 库。

最短路径的 4 个常用算法是 Floyd、Bellman-Ford、SPFA、Dijkstra。在不同的应用场景下,用户应该有选择地使用它们:

(1) 图的规模小,用 Floyd。如果边的权值有负数,需要判断负圈。

(2) 图的规模大,且边的权值非负,用 Dijkstra。

(3) 图的规模大,且边的权值有负数,用 SPFA。需要判断负圈。

再具体一点,可以总结出表 10.1。

表 10.1 对比 4 种常用算法

结点 n、边 m	边权值	选用算法	数据结构
$n<200$	允许有负	Floyd	邻接矩阵
$n\times m<10^7$	允许有负	Bellman-Ford	邻接表
更大	有负	SPFA	邻接表、前向星
	无负数	Dijkstra	邻接表、前向星

本节后面的讲解都以基础题 hdu 2544 为例，讲解不同算法的思想，并给出模板代码。

> **hdu 2544 “最短路径”**
>
> 把衣服从商店运到赛场，寻找从商店到赛场的最短路径线。
>
> 有 N 个路口，标号为 1 的路口是商店所在地，标号为 N 的路口是赛场所在地。有 M 条路，每条路的数据包括 3 个整数 A、B、C，表示路口 A 与路口 B 之间有一条路，需要 C 分钟的时间走过这条路。

作为预习，读者可以尝试用 DFS 做这一题，暴力搜索出所有可能的路径。在编程时注意用剪枝技术进行优化，如果新路径搜到一半已经比以前得到的最短路径更长，就停止搜这个路径，重新开始搜下一个。

10.9.1 Floyd-Warshall

1. 所有点对间的最短路径

如何一次性求所有结点之间的最短距离？Floyd 可以完成这一工作，其他 3 种算法都不行。而且 Floyd 是最简单的最短路径算法，程序比暴力的 DFS 更简单。需要提醒的是，Floyd 的复杂度很高，只能用于小规模的图。

Floyd 用到了**动态规划**的思想：求两点 i、j 之间的最短距离，可以分两种情况考虑，即经过图中某个点 k 的路径和不经过点 k 的路径，取两者中的最短路径。

动态规划的过程可以描述为：

(1) 令 $k=1$，计算所有结点之间(经过结点 1、不经过结点 1)的最短路径。

(2) 令 $k=2$，计算所有结点之间(经过结点 2、不经过结点 2)的最短路径，这一次计算利用了 $k=1$ 时的计算结果。

(3) 令 $k=3$，……

读者可以这样想象这个过程：

(1) 图上有 n 个结点，m 条边。

(2) 把图上的每个点看成一个灯，初始时灯都是灭的，大部分结点之间的距离被初始化为无穷大 INF，除了 m 条边连接的那些结点以外。

(3) 从结点 $k=1$ 开始操作，想象点亮了这个灯，并以 $k=1$ 为中转点，计算和调整图上所有点之间的最短距离。很显然，对这个灯的邻居进行的计算是有效的，而对远离它的那些点的计算基本是无效的。

(4) 逐步点亮所有的灯，每次点灯，就用这个灯中转，重新计算和调整所有灯之间的最

短距离，这些计算用到了以前点灯时得到的计算结果。

(5) 灯逐渐点亮，直到图上的点全亮，计算结束。

在这个过程中，由于很多计算是无效的，所以算法的效率不高。

复杂度。在下面的程序中，函数 floyd()有 3 重循环，复杂度是 $O(n^3)$，只能用于计算规模很小的图，即 $n<200$ 的情况。

hdu 2544 的 Floyd 算法代码(邻接矩阵)

```
include < bits/stdc++.h >
using namespace std;
const int INF = 1e6;                       //路口之间的初始距离,看成无穷大,相当于断开
const int NUM = 105;
int graph[NUM][NUM];                       //邻接矩阵存图
int n, m;
void floyd() {
    int s = 1;                             //定义起点
    for(int k = 1; k <= n; k++)            //floyd()的 3 重循环
        for(int i = 1; i <= n; i++)
            if(graph[i][k]! = INF)         //一个小优化,在 hdu 1704 题中很必要
            for(int j = 1; j <= n; j++)    //请思考: 把 k 循环放到 i、j 之后行不行
                if(graph[i][j] > graph[i][k] + graph[k][j])
                   graph[i][j] = graph[i][k] + graph[k][j];
//graph[i][j] = min(graph[i][j], graph[i][k] + graph[k][j]);
//上面两句这样写也行,但是 min()比较慢,如果图大,可能会超时.读者可以试试 poj 3259
    printf(" % d\n", graph[s][n]);         //输出结果
}
int main() {
    while(~scanf(" % d % d", &n, &m)) {
                  //如果图的数据很大,不能用 cin 这种慢的输入
        if(n == 0 && m == 0)  return 0;
        for(int i = 1; i <= n; i++)        //邻接矩阵初始化
            for(int j = 1; j <= n; j++)
                graph[i][j] = INF;         //任意两点间的初始距离为无穷大
        while(m -- ) {
            int a, b, c;
            scanf(" % d % d % d", &a, &b, &c);
            graph[a][b] = graph[b][a] = c;  //邻接矩阵存图
        }
        floyd();
    }
    return 0;
}
```

Floyd 算法虽然低效，但是也有优点：

(1) 程序很简单；

(2) 可以一次求出所有结点之间的最短路径；

(3) 能处理有负权边的图。

2. 判断负圈

程序中有一个有趣的地方。在程序中，结点 i 到自己的距离 graph[i][i]并没有置初值

为 0,而是 INF,读者可能觉得很奇怪;并且在计算结束之后,graph[i][i]也不是 0,而是 graph[i][i]=graph[i][u]+…+graph[v][i],即到外面绕一圈回来的最小路径。这一点可用于判断**负圈**。

负圈是这样产生的:如果某些边的权值为负数,那么图中可能有这样的环路,环路上边的权值之和为负数,这样的环路就是负圈。每走一次这个负圈,总权值就会更小,导致陷在这个圈里出不来。

利用 Floyd 算法很容易判断负圈,只要在 floyd()中判断是否存在某个 graph[i][i]<0 就行了。因为 graph[i][i]是 i 到外面绕一圈回来的最小路径,如果小于 0,说明存在负圈。此时可置 graph[i][i]的初值为 0,这样能加快判断过程。请读者练习 poj 3259 "Wormholes"题。

3. Floyd 与邻接矩阵

上面的程序用邻接矩阵存图,实现 Floyd。邻接矩阵十分浪费空间,那么用邻接表是否会更好呢?答案是在 Floyd 算法中邻接表并不比邻接矩阵好,除非两点之间有多个边,导致不能用邻接矩阵表示。因为 Floyd 的计算过程是用动态规划求所有点之间的最短距离,必须用一个 $n\times n$ 的矩阵记录状态,空间无法节省。存图的邻接矩阵可以同时用来记录状态。

4. 打印路径

有时候题目需要打印路径,请读者练习 hdu 1385 "Minimum Transport Cost"。如果有疑问,可以先学习下面的几个算法,本书都给出了打印路径的方法。

【习题】

hdu 1599 “find the mincost route”,求最小环。

hdu 3631 “Shortest Path”,Floyd 变形。

hdu 1704 “rank”,需要在 floyd()中加一个优化:if(graph[i][k]!=INF)。

10.9.2 Bellman-Ford

1. Bellman-Ford 算法

Bellman-Ford[①] 算法用来解决单源最短路径问题:给定一个起点 s,求它到图中所有 n 个结点的最短路径。

Bellman-Ford 算法的特点是只对相邻结点进行计算,可以避免 Floyd 那种大撒网式的无效计算,大大提高了效率。为理解这个算法,可以想象图上的每个点都站着一个人,初始时,所有人到 s 的距离设为 INF,即无限大。用下面的步骤求最短路径:

(1) 第一轮,给所有的 n 个人每人一次机会,问他的邻居到 s 的最短距离是多少?如果他的邻居到 s 的距离不是 INF,他就能借道这个邻居到 s 去,并且把自己原来的 INF 更新为较短的距离。显然,开始的时候,起点 s 的直连邻居(例如 u)肯定能更新距离,而 u 的邻居(例如 v),如果在 u 更新之后问 v,那么 v 有机会更新,否则就只能保持 INF 不变。特别地,在第一轮更新中,存在一个与 s 最近的邻居 t,t 到 s 的直连距离就是全图中 t 到 s 的最短距

① Bellman-Ford 的历史与改进:https://en.wikipedia.org/wiki/Bellman-Ford_algorithm(短网址:t.cn/RSrredV)。

离。因为它通过别的邻居绕路到 s，肯定更远。t 的最短距离已经得到，后面不会再更新。在很多教材中把一轮更新称为一次“**松弛**(**relax**)”，这个概念也用在 Dijkstra 算法中。

(2) 第二轮，重复第一轮的操作，再给每个人一次问邻居的机会。这一轮操作之后，至少存在一个 s 或 t 的邻居 v，可以算出它到 s 的最短距离。v 要么与 s 直连，要么是通过 t 到达 s 的。v 的最短距离也得到了，后面不会再更新。

(3) 第三轮，再给每个人一次机会……

继续以上操作，直到所有人都不能再更新最短距离为止。

一共需要几轮操作呢？每一轮操作都至少有一个新的结点得到了到 s 的最短路径。所以，最多只需要 n 轮操作就能完成 n 个结点。在每一轮操作中，需要检查所有 m 个边，更新最短距离。根据以上分析，**Bellman-Ford 算法的复杂度是 $O(nm)$**。

以上过程，每个结点可以独立进行计算，所以这个算法符合并行计算的思想，可以用在并行计算上。例如计算机网络的 BGP 路由协议，每个路由器是一个结点，它根据与邻居的信息交换，独自计算到网络中其他路由器的最短距离。BGP 是 Bellman-Ford 算法(更准确地说，是下面的 SPFA 算法)的一个典型应用。

Bellman-Ford 有现实的模型，即问路。每个十字路口站着一个警察；在某个路口，路人问一个警察，怎么走到 s 最近？如果这个警察不知道，他会问相邻几个路口的警察：“从你这个路口走，能到 s 吗？有多远？”这些警察可能也不知道，他们会继续问新的邻居。这样传递下去，最后肯定有个警察是 s 路口的警察，他会把 s 的信息返回给他的邻居，邻居再返回给邻居。最后所有的警察都知道怎么走到 s，而且是最短的路：从 s 返回信息到所有其他点的过程就像在一个平静的池塘中从 s 丢下一个石头，荡起的涟漪一圈圈向外扩散，这一圈圈涟漪经过的路径肯定是最短的。



视频讲解

问路模型里有趣的一点，并且能体现 Bellman-Ford 思想的是警察并不需要知道到 s 的完整的路径，他只需要知道从自己的路口出发往哪个方向走能到达 s，并且路最近。

下面是 hdu 2544 的 Bellman-Ford 程序，用 bellman()替换上一节的 floyd()即可。

hdu 2544 的 Bellman-Ford 算法代码(邻接矩阵)

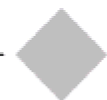
```
void bellman(){
    int s = 1;                                  //定义起点
    int d[NUM];                                 //d[i]记录结点 i 到起点 s 的最短距离.本题 s = 1
    for(int i = 1; i <= n; i++)
        d[i] = INF;                             //所有结点到 s 的距离初始化为无穷大
    d[s] = 0;                                   //以上是初始化 d[]
    for(int k = 1; k <= n; k++)                 //n 轮操作
        for(int i = 1; i <= n; i++)
            //i 和 j: 处理图中存在的边,即 graph[i][j]不等于 INF 的边
        for(int j = 1; j <= n; j++)
          if(d[j] > d[i] + graph[i][j])
                    //j 通过 i 到达起点 s: 如果距离更短,更新
            d[j] = d[i] + graph[i][j];
    printf("%d\n",d[n]);                        //输出结果
}
```

但是上面的代码并没有实用价值。由于使用了邻接矩阵这种不合适的数据结构，它没有发挥出 Bellman-Ford 的威力。Bellman-Ford 的每一轮操作只需要检查存在的 m 条边。在 $n\times n$ 的邻接矩阵中，这 m 条边是那些不等于 INF 的边，但是上面的程序却不得不检查所有 $n\times n$ 条边。

下面的程序对存储进行了优化，用 struct edge e[10005]数组来存 m 条边，避免了存储那些不存在的边。这种简单的存储方法不是邻接表，不能快速搜一个结点的所有邻居，不过正适合 Bellman-Ford 这种简单的算法。

hdu 2544 的 Bellman-Ford 算法代码(数组存边)

```
include <bits/stdc++.h>
using namespace std;
const int INF = 1e6;
const int NUM = 105;
struct edge { int u, v, w; } e[10005];          //边: 起点 u,终点 v,权值 w
int n, m, cnt;
int pre[NUM];
    //记录前驱结点.pre[x] = y,在最短路径上,结点 x 的前一个结点是 y
void print_path(int s, int t) {                  //打印从 s 到 t 的最短路径
   if(s == t){ printf("%d ", s); return; }       //打印起点
   print_path(s, pre[t]);                        //先打印前一个点
   printf("%d ", t);                             //后打印当前点.最后打印的是终点 t
}
void bellman(){
    int s = 1;                                   //定义起点
    int d[NUM];                                  //d[i]记录第 i 个结点到起点 s 的最短距离
    for (int i = 1; i <= n; i++)     d[i] = INF; //初始化为无穷大
    d[s] = 0;
    for (int k = 1; k <= n; k++)                 //一共有 n 轮操作
        for (int i = 0; i < cnt; i++){           //检查每条边
            int x = e[i].u,  y = e[i].v;
            if (d[x] > d[y] + e[i].w){
                    //x 通过 y 到达起点 s: 如果距离更短,更新
                d[x] = d[y] + e[i].w;
                pre[x] = y;                      //如果有需要,记录路径
            }
          }
        printf("%d\n", d[n]);
        //print_path(s,n);                       //如果有需要,打印路径
}
int main() {
    while(~scanf("%d%d", &n, &m)) {
        if(n == 0 && m == 0) return 0;
        cnt = 0;                                 //记录边的数量.本题的边是双向的,共有 2m 条
        while (m--) {
            int a,b,c;
            scanf("%d%d%d",&a,&b,&c);
            e[cnt].u = a;  e[cnt].v = b;  e[cnt].w = c;  cnt++;
            e[cnt].u = b;  e[cnt].v = a;  e[cnt].w = c;  cnt++;
```



```
        }
        bellman();
    }
    return 0;
}
```

2. 打印最短路径

计算出最短距离后，如果要打印整个路径，十分容易。

对于单源最短路径算法 Bellman-Ford（以及后面讲到的 Dijkstra），在连通图中，从起点 s 到任意一个结点 t 都有一条最短路径（如果有多条最短路径，就简单地选其中一条，其他的丢弃）；反过来看，从任意一个结点 t 往前追溯，沿着最短路径，一个结点一个结点往回走，就能到达起点 s。所以，只要在每个结点上记录它的前驱结点就行了。

定义 pre[]记录前驱结点。pre[x]=y 的意思是在最短路径上结点 x 的前一个结点是 y。然后用 print_path()打印整个路径。

3. 判断负圈

Bellman-Ford 也能判断负圈。当没有负圈时，只需要 n 轮就结束。如果超过 n 轮，最短路径还有变化，那么肯定有负圈。

判断负圈的程序可以写在两个 for 循环结束后。检查所有的边，如果存在某个边 (u,v)，有 $d(u)>d(v)+w(u,v)$，说明 $d(u)$的更新未结束，还能更新为更小的值。这只能是负圈引起的。

更紧凑的程序可以这样写：在循环内部判断是不是超过了 n 轮。程序如下：

hdu 2544 的 Bellman-Ford 算法代码（有判断负圈的功能）

```
void bellman(){
        int d[NUM];
        for (int i = 2;i <= n;i++)
           d[i] = INF;
        d[1] = 0;
        int k = 0;                                  //记录有几轮操作
        bool update = true;                         //判断是否有更新
        while(update) {
           k++;
           update = false;
           if(k > n) {printf("有负圈"); return;}      //有负圈,停止
           for (int i = 0; i < cnt; i++){
               int x = e[i].u, y = e[i].v;
               if (d[x] > d[y] + e[i].w){
                  update = true;
                  d[x] = d[y] + e[i].w;
               }
           }
        }
        printf("%d\n",d[n]);
}
```

10.9.3 SPFA

用队列处理 Bellman-Ford 算法可以很好地优化，这种方法叫作 SPFA。SPFA 的效率很高，在算法竞赛中的应用很广泛。

Bellman-Ford 算法有很多低效或无效的操作。分析 Bellman-Ford 算法，其核心部分是在每一轮操作中更新**所有**结点到起点 s 的最短距离。根据前面的讨论可知，计算和调整一个结点 u 到 s 的最短距离后，如果紧接着调整 u 的**邻居**结点，这些邻居肯定有新的计算结果；而如果漫无目的地计算不与 u 相邻的结点，很可能毫无变化，所以这些操作是低效的。

因此，在计算结点 u 之后，下一步只计算和调整它的邻居，这样能加快收敛的过程。这些步骤可以用队列进行操作，这就是 SPFA。

SPFA 很像 BFS：

(1) 起点 s 入队，计算它所有邻居到 s 的最短距离(当前最短距离，不是全局最短距离。在下文中，把计算一个结点到起点 s 的最短路径简称为更新状态。最后的“状态”就是 SPFA 的计算结果)。把 s 出队，状态有更新的邻居入队，没更新的不入队。也就是说，队列中都是状态有变化的结点，只有这些结点才会影响最短路径的计算。

(2) 现在队列的头部是 s 的一个邻居 u。弹出 u，更新其所有邻居的状态，把其中有状态变化的邻居入队列。

(3) 这里有一个问题，弹出 u 之后，在后面的计算中 u 可能会再次更新状态(后来发现，u 借道其他结点去 s，路更近)。所以，u 可能需要重新入队列。这一点很容易做到：在处理一个新的结点 v 时，它的邻居可能就是以前处理过的 u，如果 u 的状态变化了，把 u 重新加入队列就行了。

(4) 继续以上过程，直到队列空。这也意味着所有结点的状态都不再更新。最后的状态就是到起点 s 的最短路径。

上面第(3)点决定了 SPFA 的效率。有可能只有很少结点重新进入队列，也有可能很多。这取决于图的特征，即使两个图的结点和边的数量一样，但是边的权值不同，它们的 SPFA 队列也可能差别很大。所以，**SPFA 是不稳定的**。

在比赛时，有的题目可能故意卡 SPFA 的不稳定性：如果一个题目的规模很大，并且边的权值为非负数，它很可能故意设置了不利于 SPFA 的测试数据。此时不能冒险用 SPFA，而是用下一节的 Dijkstra 算法。Dijkstra 是一种稳定的算法，一次迭代至少能找到一个结点到 s 的最短路径，最多只需要 m(边数)次迭代即可完成。

1. 基于邻接表的 SPFA

在这个程序中，存图最合适的方法是邻接表。上面第(2)步是更新 u 的所有邻居结点的状态，而邻接表可以很快地检索一个结点的所有邻居，正符合算法的需要。程序 main()输入图时，每执行一次 e[a].push_back(edge(a,b,c))，就把边(a,b)存到了结点 a 的邻接表中；在 spfa()中，执行 for(int i=0; i<e[u].size(); i++)，就检索了结点 u 的所有邻居。

hdu 2544 的 SPFA 算法代码(邻接表+队列)

```
include <bits/stdc++.h>
```

```
using namespace std;
const int INF = 1e6;
const int NUM = 105;
struct edge{
   int from, to, w;
//边：起点 from,终点 to,权值 w.from 并没有用到,e[i]的 i 就是 from
   edge(int a, int b,int c){from = a; to = b; w = c;}
};
vector < edge > e[NUM];                    //e[i]：存第 i 个结点连接的所有边
int n, m;
int pre[NUM];
 //记录前驱结点.pre[x] = y,在最短路径上,结点 x 的前一个结点是 y
void print_path(int s,int t) {             //打印从 s 到 t 的最短路径
   ;                                 //内容与 Bellman - Ford 程序中的 print_path()完全一样
}
int spfa(int s){
   int dis[NUM];                           //记录所有结点到起点的距离
   bool inq[NUM];                          //inq[i] = true 表示结点 i 在队列中
   int Neg[NUM];                           //判断负圈(Negative loop)
   memset(Neg, 0, sizeof(Neg));
   Neg[s] = 1;
   for(int i = 1;i < = n;i++) { dis[i] = INF;  inq[i] = false; }    //初始化
   dis[s] = 0;                             //起点到自己的距离是 0
   queue < int > Q;
   Q.push(s);
   inq[s] = true;                          //起点进队列
   while(!Q.empty()) {
      int u = Q.front();
      Q.pop();                             //队头出队
      inq[u] = false;
      for(int i = 0; i < e[u].size(); i++) {   //检查结点 u 的所有邻居
          int v = e[u][i].to, w = e[u][i].w;
          if (dis[u] + w < dis[v])  {
                //u 的第 i 个邻居 v,它借道 u,到 s 更近
            dis[v] = dis[u] + w;           //更新第 i 个邻居到 s 的距离
            pre[v] = u;                    //如果有需要,记录路径
            if(!inq[v]) {
      //第 i 个邻居更新状态了,但是它不在队列中,把它放进队列
                  inq[v] = true;
                  Q.push(v);
                  Neg[v]++;
                  if(Neg[v] > n) return 1;     //出现负圈
                }
            }
        }
    }
    printf("%d\n",dis[n]);
    //print_path(s,n);                     //如果有需要,打印路径
    return 0;
}
int main(){
```

```
    while(~scanf("%d%d",&n,&m)) {
        if(n == 0 && m == 0)    return 0;
        for(int i = 1; i <= n; i++)    e[i].clear();
        while(m -- ) {
            int a,b,c;
            scanf("%d%d%d", &a, &b, &c);
            e[a].push_back(edge(a,b,c));
                    //结点 a 的邻居,都放在 node[a]里
            e[b].push_back(edge(b,a,c));
        }
        spfa(1);                              //起点是 1
    }
    return 0;
}
```

前面在讲 Bellman-Ford 的时候,曾提到它适合并行计算。读者可以发现,SPFA 比 Bellman-Ford 能更有效率地进行并行计算。例如前面提到的问路的例子,每个警察只需要在某个邻居警察通知有路径变化之后才进行计算,并把变化传递给别的邻居;如果没有收到邻居发来的变化信息,警察不需要做任何动作。这正是 SPFA 的思想。

判断负圈。SPFA 也适用于有负权值的图,也能判断负圈。如果有一个点进队列超过 n 次,那就说明图中存在负圈。具体见程序中与 Neg[]有关的部分。

打印最短路径。和前面 Bellman-Ford 打印最短路径非常相似。定义 pre[]记录前驱结点,然后用 print_path()打印整个路径。具体内容见程序。

2. 基于链式前向星的 SPFA

上面的基于邻接表的代码已经很好了,不过,在极端的情况下,图特别大,用邻接表也会超空间限制,此时就需要用到前面提到的链式前向星来存图。

建议读者认真消化下面的代码,内容包括链式前向星存图、SPFA 算法、打印最短距离、打印路径、判断负圈。这是本书精心整理的一套模板。

读者可以套用这个模板,试试 hdu 1535 "Invitation Cards"题。hdu 1535 题的图有 100 万个点,如果不用链式前向星,用别的数据结构很容易发生 MLE 错误。

hdu 2544 的 SPFA 算法代码(链式前向星)

```
include <bits/stdc++.h>
using namespace std;
const int INF = INT_MAX / 10;
const int NUM = 1000005;           //一百万个点,一百万个边
struct Edge{                       //边: edge[i]的 i 就是起点,终点 to,权值 w.下一个边 next
    int to, next, w;
}edge[NUM];
int n, m, cnt;
int head[NUM];
int dis[NUM];                      //记录所有结点到起点的距离
bool inq[NUM];                     //inq[i] = true 表示结点 i 在队列中
int Neg[NUM];                      //判断负圈(Negative loop)
int pre[NUM];                      //记录前驱结点
```

```
void print_path(int s,int t) {          //打印从 s 到 t 的最短路径
    ;                                    //内容与 Bellman-Ford 程序中的 print_path()完全一样
}
void init(){
    for(int i = 0; i < NUM; ++i){
        edge[i].next = -1;
        head[i] = -1;
    }
    cnt = 0;
}
void addedge(int u, int v, int w){      //前向星存图
    edge[cnt].to = v;
    edge[cnt].w = w;
    edge[cnt].next = head[u];
    head[u] = cnt++;
}
int spfa(int s) {
    memset(Neg, 0, sizeof(Neg));
    Neg[s] = 1;
    for(int i=1; i<=n; i++) { dis[i]=INF;  inq[i]=false; }//初始化
    dis[s] = 0;                                        //起点到自己的距离是 0
    queue<int> Q;
    Q.push(s);
    inq[s] = true;                                     //起点进队列

    while(!Q.empty()) {
        int u = Q.front(); Q.pop();                    //队头出队
        inq[u] = false;
        for(int i=head[u]; ~i; i = edge[i].next) {     //~i 也可以写成 i!=-1
            int v = edge[i].to, w = edge[i].w;
            if (dis[u]+w < dis[v]) {
                            //u 的第 i 个邻居 v,它借道 u,到 s 更近
                dis[v] = dis[u]+w;                     //更新第 i 个邻居到 s 的距离
                pre[v] = u;                            //如果有需要,记录路径
                if(!inq[v]) {
                  //邻居 v 更新状态了,但是它不在队列中,把它放进队列
                  inq[v] = true;
                  Q.push(v);
                  Neg[v]++;
                  if(Neg[v] > n) return 1;             //出现负圈
                }
            }
        }
    }
    printf("%d\n",dis[n]);                             //从 s 到 n 的最短距离
    //print_path(s,n);                                 //如果有需要,打印路径
    return 0;
}
int main() {
    while(~scanf("%d%d",&n,&m)) {
        init();
```

```
        if(n == 0 && m == 0) return 0;
        while(m -- ) {
            int u,v,w;
            scanf("%d%d%d", &u, &v, &w);
            addedge(u,v,w);
            addedge(v,u,w);
        }
        spfa(1);
    }
    return 0;
}
```

10.9.4 Dijkstra

Dijkstra 算法也用来解决单源最短路径问题。Dijkstra 是非常高效而且稳定的算法，它比前面提到的最短路径算法都复杂一些，下面先介绍它的思想。

前面在讲 Bellman-Ford 算法时，提到它在现实中的模型是找警察问路。在现实中，Dijkstra 有另外的模型，例如多米诺骨牌，读者可以想象下面的场景。

在图中所有的边上排满多米诺骨牌，相当于把骨牌看成图的边。一条边上的多米诺骨牌数量和边的权值(例如长度或费用)成正比。规定所有骨牌倒下的速度都是一样的。如果在一个结点上推倒骨牌，会导致这个结点上的所有骨牌都往后面倒下去。

在起点 s 推倒骨牌，可以观察到，从 s 开始，它连接的边上的骨牌都逐渐倒下，并到达所有能达到的结点。在某个结点 t，可能先后从不同的线路倒骨牌过来；先倒过来的骨牌，其经过的路径肯定就是从 s 到达 t 的最短路径；后倒过来的骨牌，对确定结点 t 的最短路径没有贡献，不用管它。

从整体看，这就是一个从起点 s 扩散到整个图的过程。

在这个过程中，观察所有结点的最短路径是这样得到的：

(1) 在 s 的所有直连邻居中，最近的邻居 u，骨牌首先到达。u 是第一个确定最短路径的结点。从 u 直连到 s 的路径肯定是最短的，因为如果 u 绕道别的结点到 s，必然更远。

(2) 然后，把后面骨牌的倒下分成两个部分，一部分是从 s 继续倒下到 s 的其他的直连邻居，另一部分是从 u 出发倒下到 u 的直连邻居。那么下一个到达的结点 v 必然是 s 或者 u 的一个直连邻居。v 是第二个确定最短路径的结点。

(3) 继续以上步骤，在每一次迭代过程中都能确定一个结点的最短路径。

Dijkstra 算法应用了**贪心法**的思想，即"抄近路走，肯定能找到最短路径"。

在上述步骤中可以发现：Dijkstra 的每次迭代，只需要检查上次已经确定最短路径的那些结点的邻居，检查范围很小，算法是**高效**的；每次迭代，都能得到至少一个结点的最短路径，算法是**稳定**的。

与 Bellman-Ford 对比：Bellman-Ford 是分布式的思想；而 Dijkstra 必须从起点 s 开始扩散和计算，是集中式的思想。读者可以试试在多米诺骨牌模型中运用 Bellman-Ford，看看行不行。

那么如何编程实现呢？程序的主要内容是维护两个集合，即已确定最短路径的结点集

合 A、这些结点向外扩散的邻居结点集合 B。程序逻辑如下：

(1) 把起点 s 放到 A 中，把 s 所有的邻居放到 B 中。此时，邻居到 s 的距离就是直连距离。

(2) 从 B 中找出距离起点 s 最短的结点 u，放到 A 中。

(3) 把 u 所有的新邻居放到 B 中。显然，u 的每一条边都连接了一个邻居，每个新邻居都要加进去。其中 u 的一个新邻居 v，它到 s 的距离 $\text{dis}(s,v)$ 等于 $\text{dis}(s,u)+\text{dis}(u,v)$。

(4) 重复(2)、(3)，直到 B 为空时结束。

计算结束后，可以得到从起点 s 到其他所有点的最短距离。

下面举例说明，如图 10.24 所示。

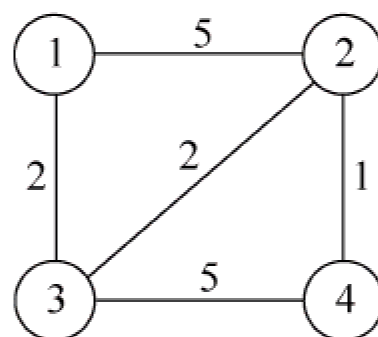


图 10.24 无向图

在图 10.24 中，起点是 1，求 1 到其他所有结点的最短路径。

(1) 1 到自己的距离最短，把 1 放到集合 A 里：$A=\{1\}$。把 1 的邻居放到集合 B 里：$B=\{(2-5),(3-2)\}$。其中(2—5)表示结点 2 到起点的距离是 5。

(2) 从 B 中找到离集合 A 最近的结点，是结点 3。在 A 中加上 3，现在 $A=\{1,3\}$，也就是说得到了从 1 到 3 的最短距离；从 B 中拿走(3—2)，现在 $B=\{(2-5)\}$。

(3) 对结点 3 的每条边，扩展它的新邻居，放到 B 中。3 的新邻居是 2 和 4，那么 $B=\{(2-5),(2-4),(4-7)\}$。其中(2—4)是指新邻居 2 通过 3 到起点 1，距离是 4。由于(2—4)比(2—5)更好，丢弃(2—5)，$B=\{(2-4),(4-7)\}$。

(4) 重复步骤(2)、(3)。从 B 中找到离起点最近的结点，是结点 2。在 A 中加上 2，并从 B 中拿走(2—4)；扩展 2 的邻居放到 B 中。现在 $A=\{1,3,2\}$，$B=\{(4-7),(4-5)\}$。由于(4—5)比(4—7)更好，丢弃(4—7)，$B=\{(4-5)\}$。

(5) 从 B 中找到离起点最近的结点，是结点 4。在 A 中加上 4，并从 B 中拿走(4—5)。此时已经没有新邻居可以扩展。现在 $A=\{1,3,2,4\}$，B 为空，结束。

下面讨论上述步骤的复杂度。图的边共有 m 个，需要往集合 B 中扩展 m 次。在每次扩展后，需要找集合 B 中距离起点最小的结点。集合 B 最多可能有 n 个结点。把问题抽象为每次往集合 B 中放一个数据，在 B 中的 n 个数中找最小值，如何快速完成？如果往 B 中放数据是乱放，找最小值也是用类似冒泡的简单方法，复杂度是 n，那么总复杂度是 $O(nm)$，和 Bellman-Ford 一样。

上述方法可以改进，得到更好的复杂度。改进的方法如下：

(1) 每次往 B 中放新数据时按从小到大的顺序放，用二分法的思路，复杂度是 $O(\log_2 n)$，保证最小的数总在最前面。

(2) 找最小值，直接取 B 的第一个数，复杂度是 $O(1)$。

此时 Dijkstra 算法总的复杂度是 $O(m\log_2 n)$，是最高效的最短路径算法。

在编程时，一般不用自己写上面的程序，直接用 STL 的优先队列就行了，完成数据的插入和提取。

下面的程序代码中有两个关键技术：

(1) 用邻接表存图和查找邻居。对邻居的查找和扩展是通过动态数组 vector < edge > e[NUM]实现的邻接表，和上一节的 SPFA 一样。其中 $e[i]$存储第 i 个结点上所有的边，边的一头是它的邻居，即 struct edge 的参数 to。在需要扩展结点 i 的邻居的时候，查找 $e[i]$即可。

已经放到集合 A 中的结点不要扩展；程序中用 bool done[NUM]记录集合 A，当 done[i]=true 时，表示它在集合 A 中，已经找到了最短路径。

(2) 在集合 B 中找距离起点最短的结点。直接用 STL 的优先队列实现，在程序中是 priority_queue <s_node> Q。但是有关丢弃的动作，STL 的优先队列无法做到。例如步骤(3)中，需要在 B={(2—5)，(2—4)，(4—7)}中丢弃(2—5)，但是 STL 没有这种操作。在程序中也是用 bool done[NUM]协助解决这个问题。从优先队列 pop 出(2—4)时，记录 done[2]=true，表示结点 2 已经处理好。下次从优先队列 pop 出(2—5)时，判断 done[2]是 true，丢弃。

下面是模板代码。

hdu 2544 的 Dijkstra 算法代码(邻接表+优先队列)

```
include<bits/stdc++.h>
using namespace std;
const int INF = 1e6;
const int NUM = 105;
struct edge{
    int from, to, w;
//边：起点,终点,权值.起点 from 并没有用到,e[i]的 i 就是 from
    edge(int a, int b,int c){from = a; to = b; w = c;}
};
vector<edge> e[NUM];                //用于存储图
struct s_node{
    int id, n_dis;                  //id：结点；n_dis：这个结点到起点的距离
    s_node(int b,int c){id = b; n_dis = c;}
    bool operator < (const s_node & a) const
    { return n_dis > a.n_dis;}
};
int n,m;
int pre[NUM];                       //记录前驱结点
void print_path(int s,int t)  {     //打印从 s 到 t 的最短路径
    ;                               //内容与 Bellman - Ford 程序中的 print_path()完全一样
}
void dijkstra(){
    int s = 1;                      //起点 s 是 1
    int  dis[NUM];                  //记录所有结点到起点的距离
    bool done[NUM];                 //done[i] = true 表示到结点 i 的最短路径已经找到
    for (int i = 1;i<= n;i++) {dis[i] = INF; done[i] = false; }  //初始化
    dis[s] = 0;                     //起点到自己的距离是 0
    priority_queue<s_node> Q;       //优先队列,存结点信息
    Q.push(s_node(s, dis[s]));      //起点进队列
    while (!Q.empty()  {
        s_node u = Q.top();         //pop 出距起点 s 距离最小的结点 u
        Q.pop();
        if(done[u.id])
            //丢弃已经找到最短路径的结点,即集合 A 中的结点
          continue;
        done[u.id] = true;
        for (int i = 0; i< e[u.id].size(); i++) {     //检查结点 u 的所有邻居
```

```
            edge y = e[u.id][i];                    //u.id 的第 i 个邻居是 y.to
            if(done[y.to])                          //丢弃已经找到最短路径的邻居结点
                continue;
            if (dis[y.to] > y.w + u.n_dis) {
                dis[y.to] = y.w + u.n_dis;
                Q.push(s_node(y.to, dis[y.to]));
                                                    //扩展新的邻居,放到优先队列中
                pre[y.to] = u.id;                   //如果有需要,记录路径
            }
        }
    }
    printf("%d\n", dis[n]);
    //print_path(s,n);                              //如果有需要,打印路径
}
int main(){
    while(~scanf("%d%d",&n,&m)) {
        if(n == 0 && m == 0) return 0;
        for (int i = 1;i <= n;i++)
            e[i].clear();
        while (m -- ) {
              int a,b,c;
            scanf("%d%d%d",&a,&b,&c);
            e[a].push_back(edge(a,b,c));
                   //结点 a 的邻居,都放在 node[a]里
            e[b].push_back(edge(b,a,c));
        }
        dijkstra();
    }
}
```

打印最短路径。和前面的 Bellman-Ford 算法一样,Dijkstra 打印最短路径也非常容易,原理和 Bellman-Ford 完全一样。首先定义 pre[]记录前驱结点,然后用 print_path()打印整个路径。具体内容见程序。

链式前向星。当图十分巨大时,需要用链式前向星存图。请读者自己总结模板,并做 hdu 1535 题。

【习题】

最短路径的题目很多,下面列出了一些训练题。

poj 1860/3259/1062/3037/3615/1511/3159(把差分约束转换为最短路径)。

hdu 1874/1596/2433/2680/4889/4568(最短路径+状态压缩 DP)。

10.10 最小生成树

最小生成树是无向图中的一个问题,也很常见。

在无向图中,连通而且不含有圈(环路)的图称为树。最小生成树(Minimal Spanning Tree,MST)的基本模型可以用下面的题目描述:

hdu 1233“还是畅通工程”

有 n 个村庄需要修通道路,已知每两个村庄之间的距离,问怎么修路,使得所有村庄都连通(但不一定有直接的公路相连,只要能间接通过公路到达即可),并且道路总长度最小?请计算最小的公路总长度。

图的两个基本元素是点和边,与此对应,有两种方法可以构造最小生成树 T。这两种算法都基于贪心法,因为 MST 问题满足贪心法的“最优性原理”,即全局最优包含局部最优。prim 算法的原理是“最近的邻居一定在 MST 上”,kruskal 算法的原理是“最短的边一定在 MST 上”。

(1) prim 算法:对点进行贪心操作。从任意一个点 u 开始,把距离它最近的点 v 加入到 T 中;下一步,把距离 $\{u,v\}$ 最近的点 w 加入到 T 中;继续这个过程,直到所有点都在 T 中。

(2) kruskal 算法:对边进行贪心操作。从最短的边开始,把它加入到 T 中;在剩下的边中找最短的边,加入到 T 中;继续这个过程,直到所有边都在 T 中。

在这两个算法中,重要的问题是判断圈。最小生成树显然不应该有圈,否则就不是“最小”了。所以,在新加入一个点或者边的时候要同时判断是否形成了圈。

10.10.1 prim 算法

图 10.25 说明了 prim 算法的步骤。设最小生成树中的点的集合是 U,开始时最小生成树为空,所以 U 为空。

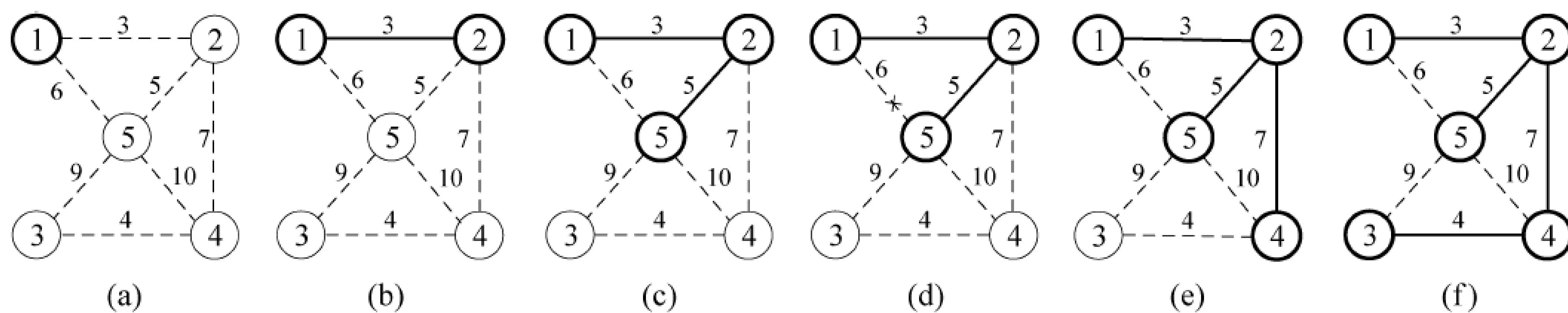


图 10.25 prim 算法

(1) 任取一点,例如点 1,放到 U 中,$U=\{1\}$,见图 10.25(a)。

(2) 找离集合 U 中的点最近的邻居,即 1 的邻居,是 2,放到 U 中,$U=\{1,2\}$,见图 10.25(b)。

(3) 找离 U 最近的点,是 5,$U=\{1,2,5\}$,见图 10.25(c)。

(4) 与 U 距离最短的是 1、5 之间的边,但是它没扩展新的点,不符合要求,见图 10.25(d)。

(5) 加入 4,$U=\{1,2,5,4\}$,见图 10.25(e)。

(6) 加入 3,$U=\{1,2,5,4,3\}$。所有点都在 U 中,结束,见图 10.25(f)。

上面的步骤和 Dijkstra 算法的步骤非常相似,不同的是 Dijkstra 需要更新 U 的所有邻居到起点的距离,即“松弛”,而 prim 不需要。所以,只要把 Dijkstra 的程序简化一些即可。

和 Dijkstra 一样,prim 程序如果用优先队列来查找距离 U 最近的点,能优化算法,此时复杂度是 $O(E\log_2 V)$。

prim 的编程比较麻烦,下面的 kruskal 算法是一种既简单又高效的算法。

10.10.2　kruskal 算法

kruskal 算法编程有以下两个关键技术：

（1）对边进行排序。可以用 STL 的 sort()函数，排序后，依次把最短的边加入到 T 中。

（2）判断圈，即处理连通性问题。这个问题用并查集简单而高效，并查集是 kruskal 算法的**绝配**。

仍以上面的图为例说明 kruskal 算法的操作步骤，如图 10.26 所示。

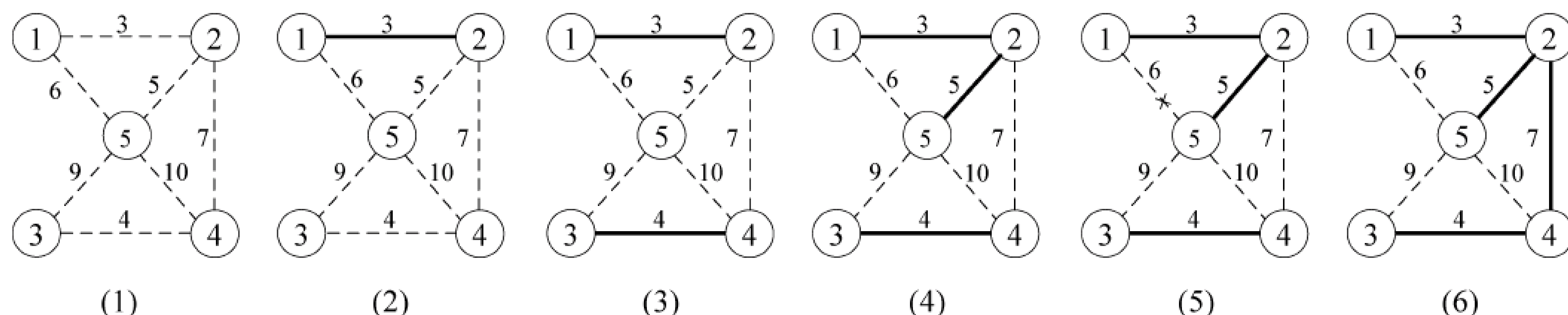


图 10.26　kruskal 算法

（1）初始时最小生成树 T 为空，见图 10.26(1)。令 S 是以结点 i 为元素的并查集，在开始的时候，每个点属于独立的集(为了便于讲解，下表中区分了结点 i 和集 S，把集的编号加上了下画线)：

S	1	2	3	4	5
i	1	2	3	4	5

（2）加入第一个最短边(1-2)：$T=\{1\text{-}2\}$，见图 10.26(2)。在并查集 S 中，把结点 2 合并到结点 1，也就是把结点 2 的集2 改成结点 1 的集1。

S	1	***1***	3	4	5
i	1	2	3	4	5

（3）加入第二个最短边(3-4)：$T=\{1\text{-}2,3\text{-}4\}$，见图 10.26(3)。在并查集 S 中，结点 4 合并到结点 3。

S	1	1	3	***3***	5
i	1	2	3	4	5

（4）加入第三个最短边(2-5)：$T=\{1\text{-}2,3\text{-}4,2\text{-}5\}$，见图 10.26(4)。在并查集 S 中，把结点 5 合并到结点 2，也就是把结点 5 的集5 改成结点 2 的集1。在集1 中，所有结点都指向了根结点，这样做能避免并查集的**长链**问题。具体原理见 5.1 节的“路径压缩”的讲解。

S	1	1	3	3	***1***
i	1	2	3	4	5

（5）第四个最短边(1-5)，见图 10.26(5)。检查并查集 S，发现 5 已经属于集1，丢弃这个边。这一步实际上是发现了一个**圈**。并查集的作用就体现在这里。

(6) 加入第五个最短边(2-4),见图 10.26(6)。在并查集 S 中,把结点 4 的集并到结点 2 的集。注意这里结点 4 原来属于集3,实际上的修改是把结点 3 的集3 改成1。

S	1	1	*1*	3	1
i	1	2	3	4	5

(7) 对所有边执行上述操作,直到结束。读者可以练习加最后两个边(3-5)、(4-5),这两个边都会形成圈。

下面是 hdu 1233 题的程序。

hdu 1233 题代码:kruskal+并查集

```
#include <bits/stdc++.h>
using namespace std;
const int NUM = 103;
int S[NUM];                                              //并查集
struct Edge {int u, v, w;} edge[NUM * NUM];              //定义边
bool cmp(Edge a, Edge b)  { return a.w < b.w; }
int find(int u) { return S[u] == u ? u : find(S[u]); }
              //查询并查集,返回 u 的根结点
int n, m;                                                //点,边
int kruskal() {
      int ans = 0;
      for(int i = 1; i <= n; i++)
          S[i] = i;                                      //初始化,开始时每个村庄都是单独的集
      sort(edge + 1, edge + 1 + m, cmp);
      for(int i = 1; i <= m; i++) {
         int b = find(edge[i].u);                        //边的前端点 u 属于哪个集
         int c = find(edge[i].v);                        //边的后端点 v 属于哪个集
         if(b == c) continue;                            //产生了圈,丢弃这个边
         S[c] = b;                                       //合并
         ans += edge[i].w;                               //计算 MST
      }
      return ans;
}
int main() {
    while(scanf("%d", &n), n) {
        m = n * (n - 1)/2;
        for(int i = 1; i <= m; i++)                      //在题目中,点的编号从 1 开始
            scanf("%d%d%d",&edge[i].u, &edge[i].v, &edge[i].w);
        printf("%d\n", kruskal());
    }
    return 0;
}
```

kruskal 算法的复杂度包括两部分,即对边的排序 $O(E\log_2 E)$、并查集的操作 $O(E)$,一共是 $O(E\log_2 E+E)$,约等于 $O(E\log_2 E)$,时间主要花在排序上。

与 prim 相比，kruskal 的编码更简单，复杂度也好，更受人们欢迎。不过，如果图的边很多，kruskal 的复杂度要差一些。简单地说，kruskal 适用于稀疏图，prim 适用于稠密图。

【习题】

最小生成树算法有一些扩展问题，例如最大生成树、次小生成树、最小瓶颈生成树等，见下面的习题。

hdu 1102，简单题。

hdu 3938，离线算法。

poj 2377，最大生成树。

hdu 5627，最大生成树。

hdu 4081，次小生成树。

hdu 4126/4756，次小生成树。用 kruskal 会超时，需要结合 prim 和树形 DP。

hdu 4750，最小瓶颈生成树。

10.11　最　大　流

最大流问题（Maximum Flow Problem）是网络流中的基本问题，它是基于有向图的。最大流问题的解决有助于解决其他网络流问题，例如最小割、二分图匹配等。

最大流问题在生活中常见的原型是水流问题。hdu 1532 题描述了这个模型。

hdu 1532 "Drainage Ditches"

约翰在农场建造了一套排水沟，以便下雨时把池塘的水排放到附近的溪流中。约翰还在每个水沟的入口安装了调节器，可以控制水流入该水沟的速度。

约翰不仅知道每个水沟每分钟可以运输多少加仑的水，而且还知道水沟的确切布局，水在这些水沟里相互进入和流动。对于任何给定的水沟，水只沿一个方向流动。水可能在某些水沟里兜圈子。

求源点 1（就是水塘）到终点 M（就是溪流）的最大流速。

在计算机网络中有带宽的概念，即每秒可传送的数据流量，和水流这个模型是一样的。

另外一个最大流模型的例子是道路的宽度。道路有单车道、双车道、四车道，同时能开行的车辆数量不同。这些不同道路的运输能力是不同的。注意这里需要假设所有车的速度都一样。

在 10.9 节中曾提到**"可加性参数"**和**"最小性参数"**。最大流问题的水流、带宽和宽度都是"最小性参数"。例如，一条路径上的最大水流由这条路径上水流容量最小的那条边决定，也就是说，由这条路径上的"瓶颈"决定。

最大流问题就是求两点间（分别称为源点、汇点）的最大流速，图中的任何点都可以作为它们的中转。在求最大流时需要满足以下 3 个性质：

（1）流量守恒。从源点 s 流出的流量和到达汇点 t 的流量相等；其他所有中转点，流入

和流出相等。

(2) 反对称性。设从 u 到 v 的流量是 $f(u,v)$,v 到 u 的流量是 $f(v,u)$,那么 $f(u,v)=-f(v,u)$。

(3) 容量限制。每个边的实际流速不大于最大流速(把最大流速称为容量)。

算法需要搜索所有的点和边。

最大流算法有很多种,基本上分为两类:

(1)"增广路"算法。例如 Edmonds-Karp 算法、Dinic 算法。

(2)"预流推进"算法。例如 ISAP 算法。

Edmonds-Karp 算法比较容易,但是效率不高,在竞赛中一般使用 Dinic 算法和 ISAP 算法。

在学习有效的最大流算法之前,读者可以自己思考暴力法或者简单的贪心法。

10.11.1 Ford-Fulkerson 方法

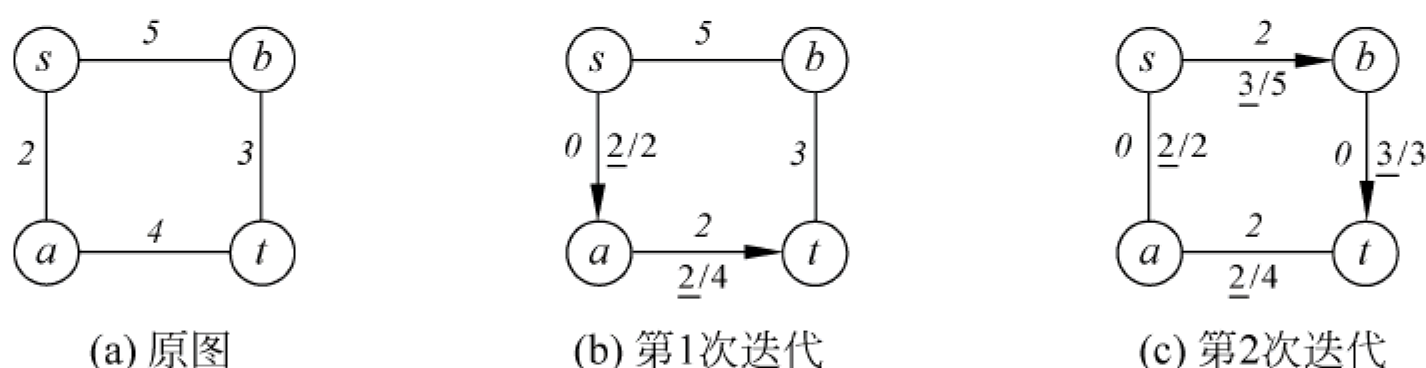
Edmonds-Karp 算法是 Ford-Fulkerson 方法的一种实现。所谓 Ford-Fulkerson 方法,是一种非常容易理解的算法思想:

(1) 在初始的时候,所有边上的流量为 0。

(2) 找到一条从 s 到 t 的路径,按 3 个性质得到这条路径上的最大流,更新每个边的残留容量。残留容量在后续步骤中继续使用。

(3) 重复步骤(2),直到找不到路径。

以图 10.27 为例[①],图(a)中的斜体数字标出了每个边的容量,开始时每个边的流量是 0;图 10.27(b)是第 1 次迭代,找到了一条路径 $s \to a \to t$,画线数字是每个边的流量,斜体数字是残留容量;图 10.27(c)是第 2 次迭代,找到了一条路径 $s \to b \to t$,更新每个边的流量和残留容量。第 2 次迭代后没有新的路径,结束(注意:这里为了介绍思想,简化了过程;这实际上是**有错误**的,解释见下面的"残留网络")。



(a) 原图　　(b) 第1次迭代　　(c) 第2次迭代

图 10.27　Ford-Fulkerson 方法示意

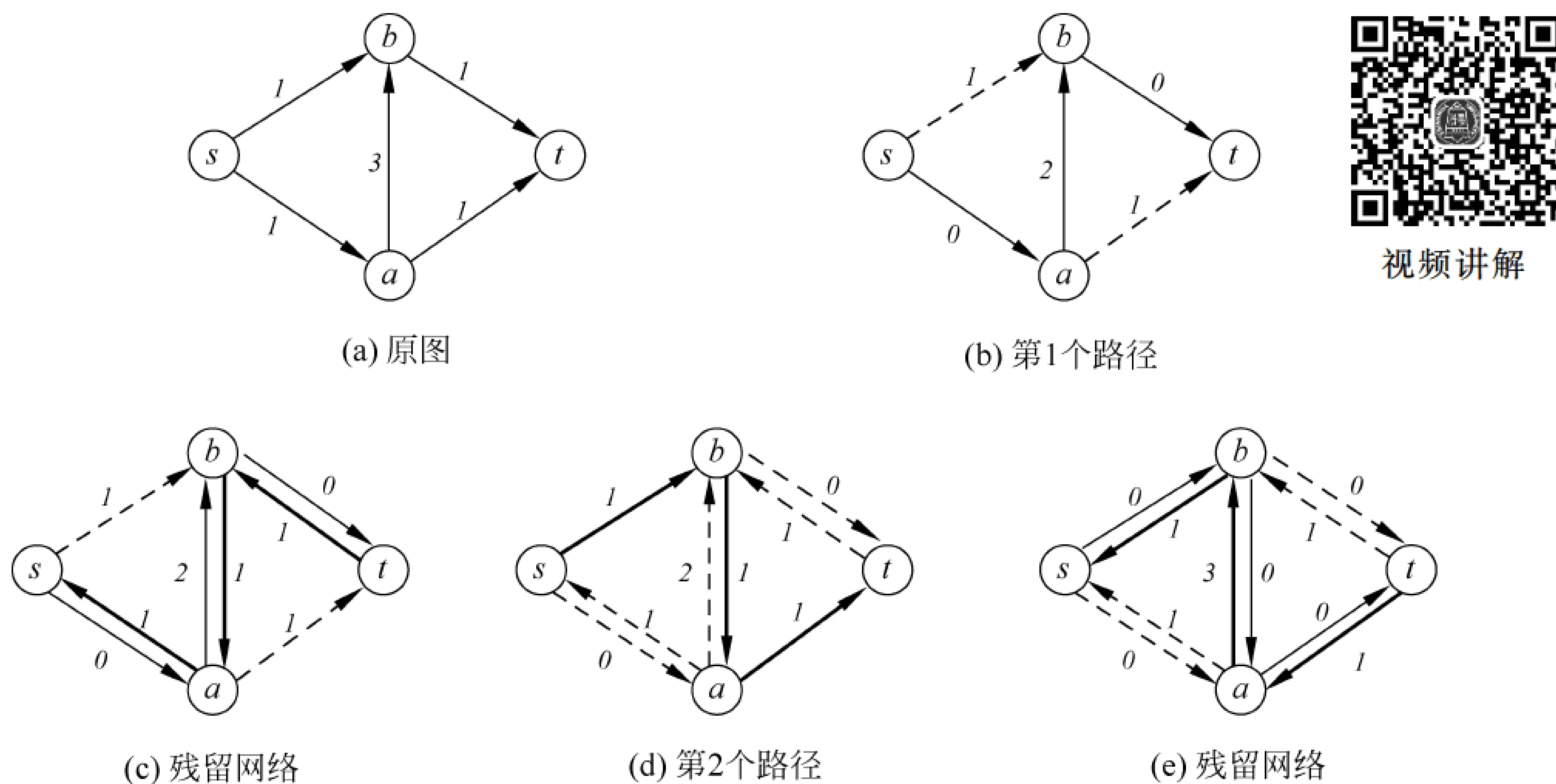
Ford-Fulkerson 方法基本上就是上述的思路。它有 3 个思想,也是后文将提到的"最大流最小割定理"的基础:

(1) 残留网络(residual network)。迭代后残留容量所产生的图,每次新的迭代在上一次的残留网络上进行。

但是,它实际上**并不是**图 10.27(a)、(b)、(c)中的斜体数字所表示的图,因为这个图在

① 这个例子过于简单,更完整的例子请参考《算法导论》,Thomas H. Cormen 等著,潘金贵等译,机械工业出版社,26.2 节,图 26-5。

迭代过程中损失了一些信息。请读者仔细分析图 10.28 所示的图解。



(a) 原图 (b) 第1个路径 (c) 残留网络 (d) 第2个路径 (e) 残留网络

图 10.28 残留网络

对于图 10.28(a)上的最大流,容易发现,在 *s-a-t*、*s-b-t* 这两条路径上最大流等于 2。

下面找一条路径。图 10.28(b)是搜到的第 1 个路径(读者可以想象 *s-b*、*a-t* 原来不存在水沟),产生的流量是 1;图上的数字是残留容量。如果在这个图上继续搜索路径,已经没有新路径。这显然是不对的。其原因是第 1 次搜索的结果影响了后续的路径搜索。那么如何消除这个影响?

图 10.28(c)是解决方法,在上一次的路径上补充反向路径,其值就是用过的流量 1,形成的新网络图就是残留网络。

残留网络的原理可以这样理解:在搜索新的增广路时,可能会经过以前的增广路使用过的水沟,而这个新路的水流可能与原来的水流相反,所以需要补上反向路径,让新的搜索有反向水流的机会。

图 10.28(d)是在图 10.28(c)的基础上搜到的第 2 个路径,这次结果是对的。

图 10.28(e)是最后的残留网络。此时,从 s 到 t,在残留网络上不存在新的路径,结束。为加深理解,请读者验证并思考:最后的残留网络,两点之间反向路径的值就是两点之间的实际流量。所以,可以利用残留网络输出最大流时各水沟中的实际流量。

残留网络和残留网络的反向路径是 Ford-Fulkerson 方法**最关键的技术**。

(2) 增广路(augmenting path)。在残留网络上找到的一条从 s 到 t 的路径。

(3) 割(cut)。Ford-Fulkerson 方法的正确性是最大流最小割定理的推论:一个流是最大流,当且仅当它的残留网络不包含增广路径时。

Ford-Fulkerson 方法的运行时间依赖于增广路径的搜索次数。虽然用 BFS 或者 DFS 都行,但是 DFS 这种深度搜索模式可能陷入长时间的迭代,图 10.29 是一个例子。

在图 10.29(b)和(c)中,很不幸地,DFS 选择了 *s-b-a-t* 和 *s-a-b-t* 这种绕路,接下来又反复选择这两个路径。在到达终点图 10.29(d)前,共迭代了约 200 次。

如果用 BFS,几次就够了。

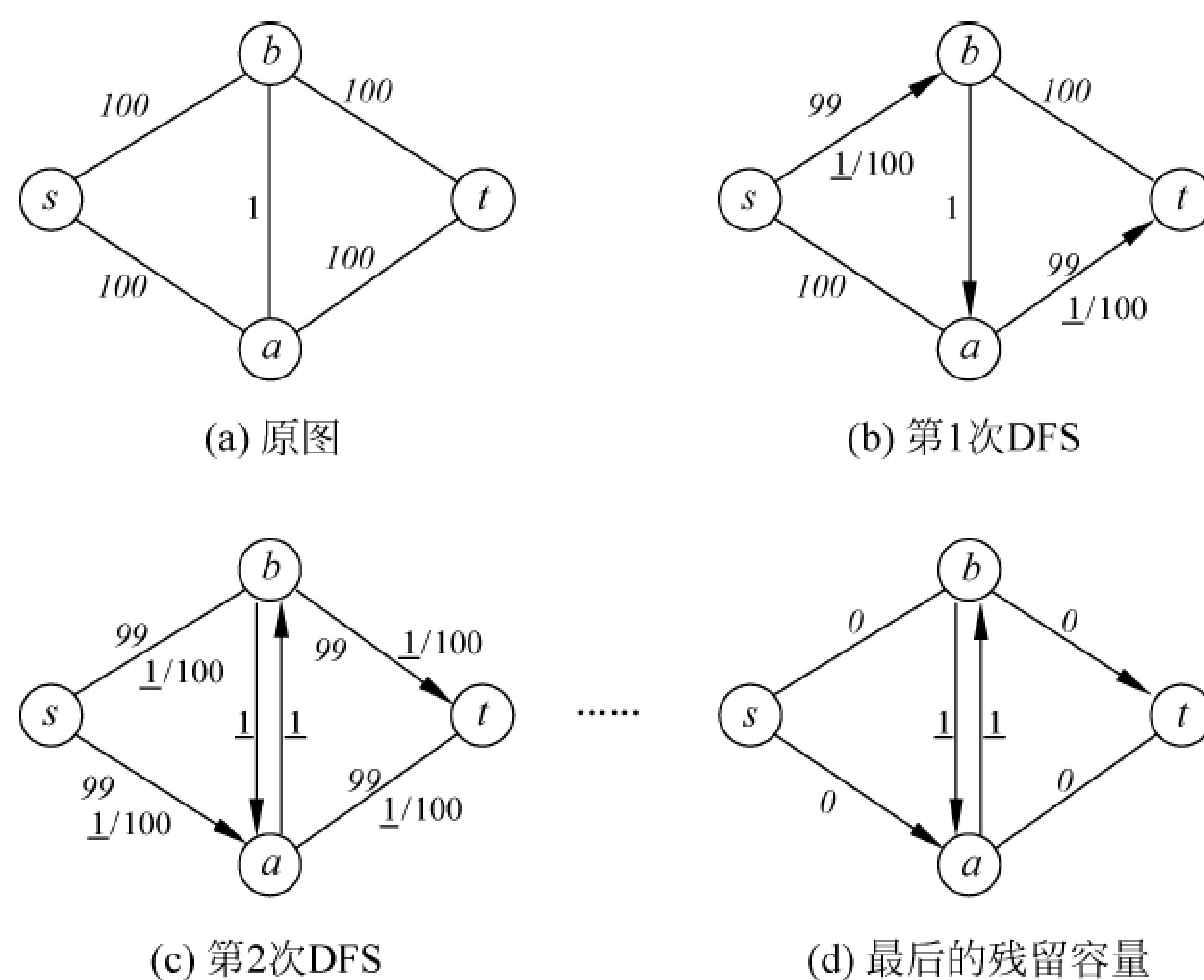


图 10.29　DFS 模式陷入长时间的迭代

10.11.2　Edmonds-Karp 算法

如果用 BFS 来计算增广路径，就是 Edmonds-Karp 算法。

复杂度：经过 $O(VE)$次 BFS 迭代，所有增广路被找到；一次 BFS 的时间是 $O(E)$，所以总时间是 $O(VE^2)$。

由于 Edmonds-Karp 算法的复杂度高，只能用于小图，所以用邻接矩阵存图就行了。

下面是 hdu 1532 题的代码，用矩阵 graph[][]存图，它同时也用于记录更新后的残留网络。

hdu 1532 题的代码

```
#include<bits/stdc++.h>
const int INF = 1e9;
const int maxn = 300;
using namespace std;
int n, m, graph[maxn][maxn], pre[maxn];
            //graph[][]不仅记录图，还是残留网络
int bfs(int s,int t){
    int flow[maxn];
    memset(pre,-1,sizeof pre);
    flow[s] = INF; pre[s] = 0;                    //初始化起点
    queue<int> Q;  Q.push(s);                     //起点入栈，开始 BFS
    while(!Q.empty()){
        int u = Q.front();  Q.pop();
        if(u==t) break;                           //搜到一个路径，这次 BFS 结束
        for(int i=1; i<=m; i++){                  //BFS 所有的点
            if(i!= s && graph[u][i]>0 && pre[i] == -1){
                pre[i] = u;                       //记录路径
                Q.push(i);
```

```
                flow[i] = min(flow[u],graph[u][i]); //更新结点流量
            }
        }
    }
    if(pre[t] == -1) return -1;                    //没有找到新的增广路
    return flow[t];                                //返回这个增广路的流量
}
int maxflow(int s, int t){
    int Maxflow = 0;
    while(1){
        int flow = bfs(s,t);
            //执行一次 BFS,找到一条路径,返回路径的流量
        if(flow == -1) break;                      //没有找到新的增广路,结束
        int cur = t;                               //更新路径上的残留网络
        while(cur!= s){                            //一直沿路径回溯到起点
            int father = pre[cur];                 //pre[]记录路径上的前一个点
            graph[father][cur] -= flow;            //更新残留网络：正向减
            graph[cur][father] += flow;            //更新残留网络：反向加
            cur = father;
        }
        Maxflow += flow;
    }
    return Maxflow;
}
int main(){
    while(~scanf("%d%d",&n,&m)){
        memset(graph,0,sizeof graph);
        for(int i=0; i<n; i++){
            int u,v,w;
            scanf("%d%d%d",&u,&v,&w);
            graph[u][v] += w;                      //可能有重边
        }
        printf("%d\n",maxflow(1,m));
    }
    return 0;
}
```

最大流的建模问题。前面最大流的模型是基于有向图的,而且只有一个源点和一个汇点,但是题目所给的条件不一定这么严格,此时需要转换为下面的模型。

(1) 无向图转换为有向图。如果给的是无向图,可以把 u、v 之间的无向边变为(u,v)、(v,u)两个有向边,容量一样。u、v 的实际流量为两者的实际流量之差,即互相**抵消**,例如从 u 到 v 的流量是 10,从 v 到 u 的流量是 4,那么从 u 到 v 的流量是 $10-4=6$。

(2) 多个源点和多个汇点。此时可以添加一个"超级源点"s 和一个"超级汇点"t。从 s 到每个源点都连一条有向边;从每个汇点都连一条边到 t。边的容量根据题目要求灵活指定。

在 10.13 节中,例题 poj 2135 "Farm Tour"就用到了这两个转换方法。在 10.14 节中有多源点、多汇点的情况。

10.11.3 Dinic 算法和 ISAP 算法

Edmonds-Karp 算法的效率低，在竞赛时若遇到规模较大的最大流问题，需要用高效的 Dinic 算法和 ISAP 算法。

Dinic 算法是对 Edmonds-Karp 算法的优化，时间复杂度理论上是 $O(V^2E)$，实际上更好，比 Edmonds-Karp 算法的 $O(VE^2)$ 强很多。

ISAP 算法的复杂度也是 $O(V^2E)$，但是比 Dinic 算法更好一些，更受欢迎。

Dinic 算法和 ISAP 算法①相当复杂，代码也比 Edmonds-Karp 算法长得多，在竞赛的时候靠自己写出来很困难。建议读者阅读有关资料，搞懂原理，学习其思想；然后找到合适的模板，特别是 ISAP 算法，学会使用它，在比赛的时候带上。

下面用最大流求解混合图的欧拉回路。

最大流算法是网络流算法的基础，它有很多应用。例如，最大流算法可用于判断和求解混合图的欧拉回路。请读者先回顾 10.5 节。

hdu 1956 "Sightseeing Tour"
给定一个图，其中同时存在有向边和无向边，问该图是否存在欧拉回路。

有向图存在欧拉回路的充要条件是所有点的度数为 0。把每个点连接的无向边改成有向边，看度数是否为 0。但是无向边很多，情况复杂，不能直接用暴力的方法做。

读者可以先思考，尝试用最大流方法解决。然后阅读下面的解题思路。

把所有的无向边任意定个方向，把这个包括原来的有向边和设定了方向的无向边的图称为初始图 G，然后计算每个点的度数。点 i 的度数 degree[i]＝出度－入度，有以下两种情况：

(1) 存在一个 degree[i]为奇数。如果把 i 的一个无向边改个方向，那么 degree[i]变为 degree[i]＋2 或 degree[i]－2，仍然是奇数，不会等于 0，所以不存在欧拉回路。

(2) 所有的 degree[i]全是偶数。可以把某个 i 的一个无向边改个方向，degree[i]变为 0。那么是否**所有的点的度数都能变为 0** 呢？可以借助最大流来判断。

下面用初始图 G 建一个新图 G'，在 G 中计算得到的 degree[i]也用于建图。首先把初始图 G 中原来的有向边删除，保留定向了的无向边。然后建一个源点 s，连接**所有的** degree[i]＞0 的点，边的容量为 degree[i]/2。建一个汇点 t，把所有 degree[i]＜0 的点连接到 t，容量为 degree[i]/2。其他 degree[i]＝0 的点就不用连接 s 和 t 了。所有没有连接 s 和 t 的边，容量都为 1。

求新网络 G' 的最大流。如果从 s 出发的所有边都满流，则存在欧拉回路。把所有的有流的边全部反向，把原图中的有向边再重新加入，就得到了一个有向欧拉回路。

上述算法正确吗？或者说，上述算法的结果能使得所有点的度数为 0 吗？

分 3 种情况观察：

① Dinic 算法和 ISAP 算法的对比：https://www.cnblogs.com/zhsl/archive/2012/12/03/2800092.html（永久网址：https://perma.cc/LAP9-QH83）。

（1）观察源点 s 所连接的点 v，是否能得到 degree(v)=0 的结果。在图 10.30 所示的例子中，图 10.30(a)是初始图 G 的局部，v 在 G 中有 4 个边，degree[v]=4，其中虚线是有向边，在 G' 中被删除了，剩下的 3 个实线边是原来的无向边，把方向定为出度。在图 10.30(b)中，加上源点 s，边(s,v)的容量是 degree[v]/2=2，v 的其他边的容量是 1。经过最大流的计算，如果(s,v)是满流 2，生成了图 10.30(c)中粗线条表示的流。把有流的边**反向**，得到图 10.30(d)，可以发现，degree(v)=0，符合欧拉回路的要求。从这个图也能理解为什么把边(s,v)的容量设定为 degree[v]/2。

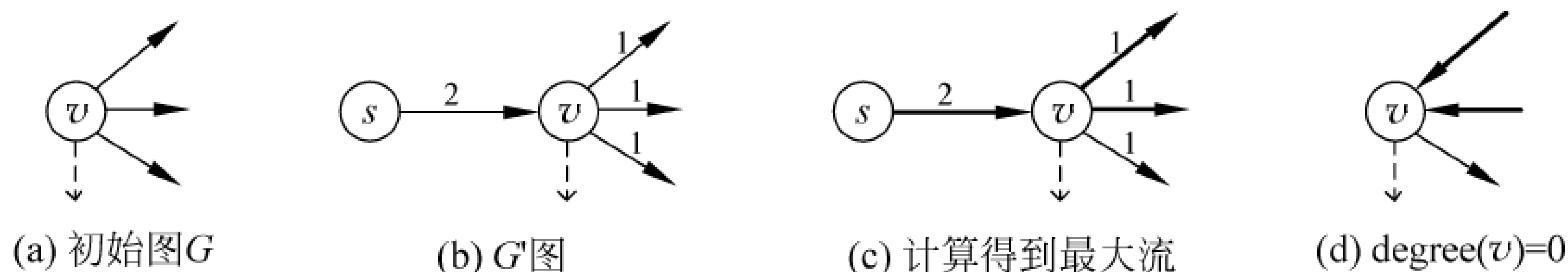


图 10.30　源点 s 连接的点 v

（2）与汇点 t 连接的点，分析同上。

（3）不与 s 和 t 连接的点 i，是否最后也有 degree(i)=0 的结果？这些点在初始图 G 中有 degree(i)=0。在 G' 中计算最大流的路径时，如果增广路经过了点 i，那么肯定有一个进边的流和一个出边的流，把这两个边同时反向，仍然是一个进边和一个出边，仍保持 degree(i)=0。

hdu 1956 的数据比较大，需要用 Dinic 或 ISAP 算法。

【习题】

hdu 3549 “Flow Problem”，最大流入门题。

hdu 4280 “Island Transport”，数据规模为 $2 \leqslant N, M \leqslant 100\ 000$。ISAP 模板题，用 Dinic 有可能超时。

hdu 3472 “HS BDC”。有 n 个单词，有的可以前后颠倒，看是否可以将 n 个单词首尾相连。混合图欧拉回路。

10.12　最　小　割

s-t 最小割是最大流的一个直接应用。

割(cut)和 s-t 割的概念：在有向图流网络 $G=(V,E)$ 中，割把图分成 S 和 $T=V-S$ 两部分，源点 $s \in S$，汇点 $t \in T$，这称为 s-t 割。

在图 10.31 中，边上的数字标出了流量和容量，s 和 t 之间的流量是 14。图中的虚线是一个割，把图分成了 S、T 两部分。

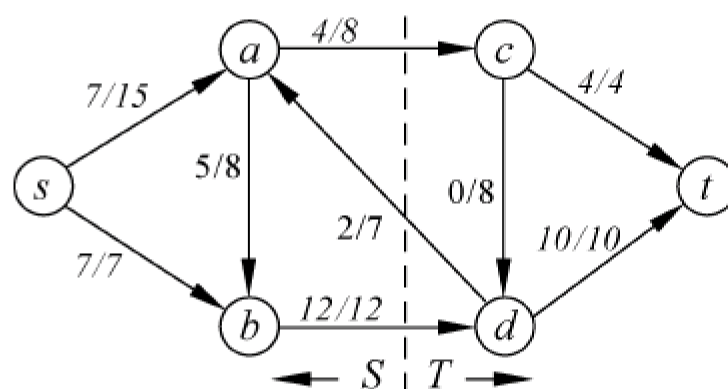


图 10.31　s-t 割

从 S 到 T，穿过割的净流量是 4+12−2=14。显然，在 s、t 之间做任意割，流经这个割的净流量都相等。

S 经过这个割到 T 的容量是 8+12=20，分别是边 ac 和 bd。也就是说，如果把边 ac 和 bd 去掉，S 中的水就

不能流到 T。注意在计算 S 到 T 的容量时不要算从 T 到 S 的反向容量。图中的虚线并不是一个最小割，读者可以观察最小割在哪里。

s-t 最小割问题是针对容量的，就是找到源点 s 和汇点 t 之间容量最小的割。

最小割问题可以形象地理解为：为了不让水从 s 流向 t，怎么破坏水沟代价最小？被破坏的水沟必然是从 s 到 t 的单向水沟。

最大流最小割定理：源点 s 和汇点 t 之间的最小割等于 s 和 t 之间的最大流[①]。

需要注意的是，定理中的最大流是指流量，而最小割是指容量。

全局最小割：把 s-t 最小割问题扩展到全局，有全局最小割问题。

简单的思路：可以利用最小割最大流定理，即枚举每个点当作汇点，计算出它的最大流，然后在所有点的最大流中取最小值。

但是这样做的复杂度很高，枚举汇点要 $O(V)$，Dinic 或 ISAP 算法的复杂度是 $O(V^2E)$，总复杂度是 $O(V^3E)$。

解决此类问题需要用 Stoer-Wagner 算法，由于题目比较罕见，本书不展开介绍。读者可以通过 poj 2914 “Minimum Cut”来了解。

【习题】

普通最小割问题的编码就是最大流算法。在对问题正确建模之后，用最大流的算法思路解决。

hdu 3251 “Being a Hero”，最小割。

poj 1815 “Friendship”，最小割。

10.13 最小费用最大流

在最大流网络中，每条边只有一个限制条件，例如容量、带宽等，这是“最小性参数”，现在加上一个新的限制条件，例如费用，这是“可加性参数”。在两个限制条件的基础上引出了最小费用最大流问题：当流量为 F 时，求费用最小的流；如果没有指定 F，就是求最大流时的最小费用。

有两种思路：

(1) 先求一个最大流，然后不断优化得到最小费用流。首先用最大流算法得到一个最大流，然后检查边的情况，看是否有费用更小同时也能满足最大流的边，如果有，就进行调整，得到一个新的最大流。经过多次迭代，直到所有边都无法调整，就得到了最小费用最大流。

(2) 从零流开始，每次增加一个最小费用路径，经过多次增广，直到无法再增加路径，就得到了最大流。

思路(2)更容易理解和操作，它是网络流问题和最短路径问题的结合，其算法也是最大流算法和最短路径算法的结合。

---

① 证明见《算法导论》，Thomas H. Cormen 等著，潘金贵等译，机械工业出版社，定理 26.7。

最短路径算法有 Bellman-Ford 算法、Dijkstra 算法等，是否都能用？如果边的费用权值有负数，只能选择 Bellman-Ford 算法（或 SPFA 算法）。在最小费用最大流算法中，由于残留网络用到了反向边，所以肯定会出现负权边[①]，在本节的例题中会说明这一问题。

最小费用最大流的解决方法是 Ford-Fulkerson 方法+Bellman-Ford 算法（SPFA 算法）。

回顾最大流的 Ford-Fulkerson 方法，它的主要操作是在残留网络上不断寻找增广路径。如果用 BFS 求增广路，就是 Edmonds-Karp 算法。BFS 求增广路是很盲目的，它不会区分增广路的“好坏”。

如何找一条“好”的增广路？如果不用 BFS，而是改用 Bellman-Ford 算法（SPFA 算法），每次在残留网络上找增广路时都找费用最小的路径，就会得到一条“好”的、费用最低的路径。不断用 Bellman-Ford 算法（SPFA 算法）求增广路，直到满足题目要求的流量 F，最后得到一个流量为 F 并且费用最小的流。

上述的算法思想是否正确？可以简单思考如下：如果经过上述步骤得到的不是最小费用流，说明在残留网络上还存在费用更小的路径，这与前面步骤中已经计算了最小路径相矛盾[②]。

算法的复杂度是多少？找一次增广路，这个路径上至少有一个流量；总流量为 F，最多需要找 F 次增广路；每次使用 Bellman-Ford 算法找增广路，一次 Bellman-Ford 的时间是 $O(VE)$，所以总时间是 $O(FVE)$。

对于下面的例题，请读者先思考，再看答案。

poj 2135 “Farm Tour”

一个无向图，有 N 个地点、M 条边。一个人从 1 号点走到 N 号点，再从 N 号点走回 1 号点，每条路只能走一次。求来回的总长度最短的路线。

输入：第 1 行是两个整数 N、M；后面有 M 行，每行有 3 个整数，描述一个边的两个端点和边的长度。$1 \leqslant N \leqslant 1000$，$1 \leqslant M \leqslant 10\,000$。

输出：来回总长度最短的路径长度。

题目的测试数据确保存在来回的不重复路径。

根据题意分析，这是个无向图，从 1 走到 N 和从 N 走到 1 是一样的，那么题目转换为从 1 号点到 N 号点至少有两条不同路线，找其中两条，使它们的总长度最短。

刚看到这一题的时候，读者可能觉得很简单：先求第一个最短路径，然后把走过的路删除，再算一次最短路径。

然而这样做是错误的。例如图 10.32，找 a 到 d 的两条路。图中确实存在两条路，但是直接算两次最短路径却找不到这两条路：第一条最短路径是 a-c-b-d，有 3 个边，如果删除这 3 个边，图就断开了，无法继续找第二条路。

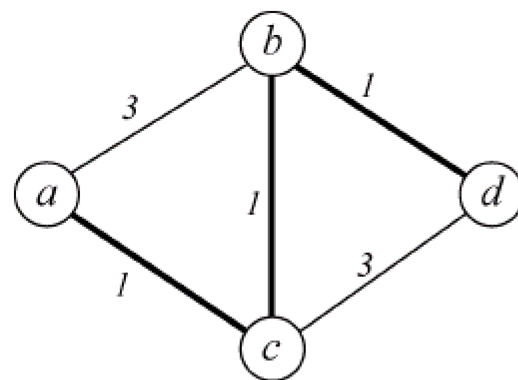


图 10.32 寻找 a 和 d 之间的两条路径

① 通过导入“势”的概念，可以在最小费用最大流算法中用 Dijkstra 算法，从而降低算法复杂度。请参考《挑战程序设计竞赛》（秋叶拓哉），225 页，“3.5.6 最小费用流”。

② 算法正确性的具体证明参考《挑战程序设计竞赛》（秋叶拓哉），225 页，“3.5.6 最小费用流”。

这个例子是从前面最大流的“残留网络”的例子引用过来的。这个例子说明本题和最大流有关系。

这一题实际上是一道最小费用最大流的裸题。建模如下：

把每条边的流量设为1，表示每条边只能用1次，把边的长度看成每个边的费用。在图中添加一个“超级源点”s和一个“超级汇点”t，s到1有一个长度为0、容量为2的边；N到t有一个长度为0、容量为2的边。在经过这个建模之后，原题中求两条最短路径的费用等价于求源点s和汇点t的最小费用最大流[①]。

分析复杂度，最小费用最大流的复杂度是$O(FVE)$，$F=2$，$V=1000$，$E=10\,000$，$FVE=$2000万，正好满足。

下面的最小费用最大流程序综合了SPFA算法和最大流算法，基本上套用了前面讲解过的模板。其中需要特别注意的是图的初始化，即如何把无向图转为有向图。

无向图的两个点(u,v)之间只有1个边，本题把它变成了4个边。

首先把无向边(u,v)分成有向边(u,v)和(v,u)。

然后把它们各分成两个边。例如有向边(u,v)变成了一个正向的费用为cost、容量为capacity的边，以及一个反向的费用为－cost、容量为0的边。这样做和最大流中的残留网络是同样的道理，相当于一次增广之后生成的残留网络。如果读者不能理解，请回顾最大流中“反向路径”的相关内容。

从这个例子可以看出，边的权值会出现负数，所以不能用Dijkstra算最短路径，只能用SPFA。

poj 2135 程序（邻接表存图＋SPFA＋最大流）

```
#include <stdio.h>
#include <algorithm>
#include <cstring>
#include <queue>
using namespace std;
const int INF = 0x3f3f3f3f;
const int N = 1010;
int dis[N], pre[N], preve[N];
//dis[i]记录起点到 i 的最短距离.pre 和 preve 见下面的注释
int n, m;
struct edge{
    int to, cost, capacity, rev;                    //rev 用于记录前驱点
    edge(int to_,int cost_,int c,int rev_){
          to = to_; cost = cost_; capacity = c; rev = rev_;}
};
vector<edge> e[N];                                  //e[i]: 存第 i 个结点连接的所有的边
void addedge(int from,int to,int cost,int capacity){  //把 1 个有向边再分为两个
    e[from].push_back(edge(to, cost, capacity, e[to].size()));
    e[to].push_back(edge(from, -cost, 0, e[from].size()-1));
}
bool spfa(int s, int t, int cnt){                   //套 SPFA 模板
```

① 从这一题的建模过程可以看出，单源最短路径问题是费用流问题的一个特殊情况。把每个边的容量设为1，添加一个源点s，s到起点的边容量是1、费用是0，那么s到终点的最小费用最大流就是最短路径。

```
    bool inq[N];
    memset(pre, -1, sizeof(pre));
    for(int i = 1; i<=cnt; ++i) { dis[i] = INF; inq[i] = false; }
    dis[s] = 0;
    queue<int> Q;
    Q.push(s);
    inq[s] = true;
    while(!Q.empty()){
        int u = Q.front();
        Q.pop();
        inq[u] = false;
        for(int i=0; i<e[u].size(); i++)
            if(e[u][i].capacity>0){
                int v = e[u][i].to, cost = e[u][i].cost;
                if(dis[u]+cost<dis[v]){
                    dis[v] = dis[u]+cost;
                    pre[v] = u;             //v 的前驱点是 u
                    preve[v] = i;           //u 的第 i 个边连接 v 点
                    if(!inq[v]){
                        inq[v] = true;
                        Q.push(v);
                    }
                }
            }
    }
    return dis[t] != INF;                   //s 到 t 的最短距离(或者最小费用)是 dis[t]
}
int mincost(int s, int t, int cnt){        //基本上是套最大流模板
    int cost = 0;
    while(spfa(s, t, cnt)){
        int v = t, flow = INF;              //每次增加的流量
        while(pre[v] != -1){                //回溯整个路径,计算路径的流
            int u = pre[v], i = preve[v];
                    //u 是 v 的前驱点,u 的第 i 个边连接 v
            flow = min(flow, e[u][i].capacity);
                    //所有边的最小容量就是这条路的流
            v = u;                          //回溯,直到源点
        }
        v = t;
        while(pre[v] != -1){                //更新残留网络
            int u = pre[v], i = preve[v];
            e[u][i].capacity -= flow;       //正向减
            e[v][e[u][i].rev].capacity += flow; //反向加,注意 rev 的作用
            v = u;                          //回退,直到源点
        }
        cost += dis[t]*flow;
                    //费用累加.如果程序需要输出最大流,在这里累加 flow
    }
    return cost;                            //返回总费用
}
int main(){
    while(~scanf("%d%d", &n, &m)){
        for(int i = 0; i<N; i++)  e[i].clear(); //清空待用
        for(int i = 1; i<=m; ++i){
```

```
            int u, v, w;
            scanf("%d%d%d", &u, &v, &w);
            addedge(u, v, w, 1);                    //把1个无向边分为2个有向边
            addedge(v, u, w, 1);
        }
        int s = n+1, t = n+2;
        addedge(s, 1, 0, 2);                        //添加源点
        addedge(n, t, 0, 2);                        //添加汇点
        printf("%d\n", mincost(s, t, n+2));
    }
    return 0;
}
```

【习题】

hdu 3376 “Matrix Again”,费用流裸题。

hdu 3667 “Transportation”。

hdu 5520 “Number Link”。

10.14 二分图匹配



视频讲解

二分图:把无向图 $G=(V, E)$分为两个集合 V_1、V_2,所有的边都在 V_1 和 V_2 之间,而 V_1 或 V_2 的内部没有边。V_1 中的一个点与 V_2 中的一个点关联,称为一个匹配。

一个图是否为二分图,一般用“染色法”进行判断。用两种颜色对所有顶点进行染色,要求一条边所连接的两个相邻顶点的颜色不相同。染色结束后,如果所有相邻顶点的颜色都不相同,它就是二分图。

一个图是二分图,当且仅当它不含边的数量为奇数的圈。读者可以画图理解这一点。

常见的二分图匹配问题有两种。

(1) 无权图,求包含边数最多的匹配,即二分图的最大匹配。本节讲解这个问题。

(2) 带权图,求边权之和尽量大的匹配。使用 KM 算法,本书没有涉及。

1. 二分图最大匹配问题

可以将二分图最大匹配问题转化为求最大流问题的思想来解决。不过在竞赛时一般不用标准的最大流模板,而是使用更简单的匈牙利算法。

二分图最大匹配的原型见下面的题目。

> **hdu 2063 “过山车”**
>
> 大家去坐过山车。过山车的每一排只有两个座位,并且必须一男一女配对坐。但是,每个女孩有各自的想法,比如 Rabbit 只愿意和 XHD 或 PQK 坐,Grass 只愿意和 linle 或 LL 坐,等等。boss 刘决定,只让能配对的人坐过山车。当然,能配对的人越多越好。问最多有多少对组合可以坐上过山车?

2. 用最大流求解二分图匹配

二分图最大匹配问题可以转化为最大流问题：把每个边都改为有向边，流量都是1；在V_1上加一个人为的源点s，它连接V_1的所有点；在V_2上加一个人为的汇点t，它连接V_2的所有点，那么s、t之间的最大流就是最大二分图匹配。

原理很直观。在图10.33中，$V_1=\{a,b,c\}$是女生，$V_2=\{x,y,z\}$是男生。例如a点，流入a的流量是1，那么从a流出的只能是1，也就是说a只能匹配$\{x,y,z\}$中的一个。从V_1到V_2的流量和从s到t的流量相等。

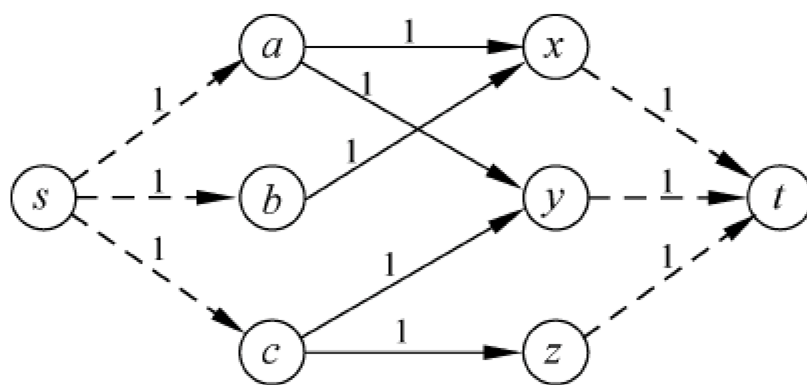


图10.33 二分图匹配和最大流

下面用最大流来求解。读者可以用图10.34复习最大流的Ford-Fulkerson方法，主要是对残留网络的操作。

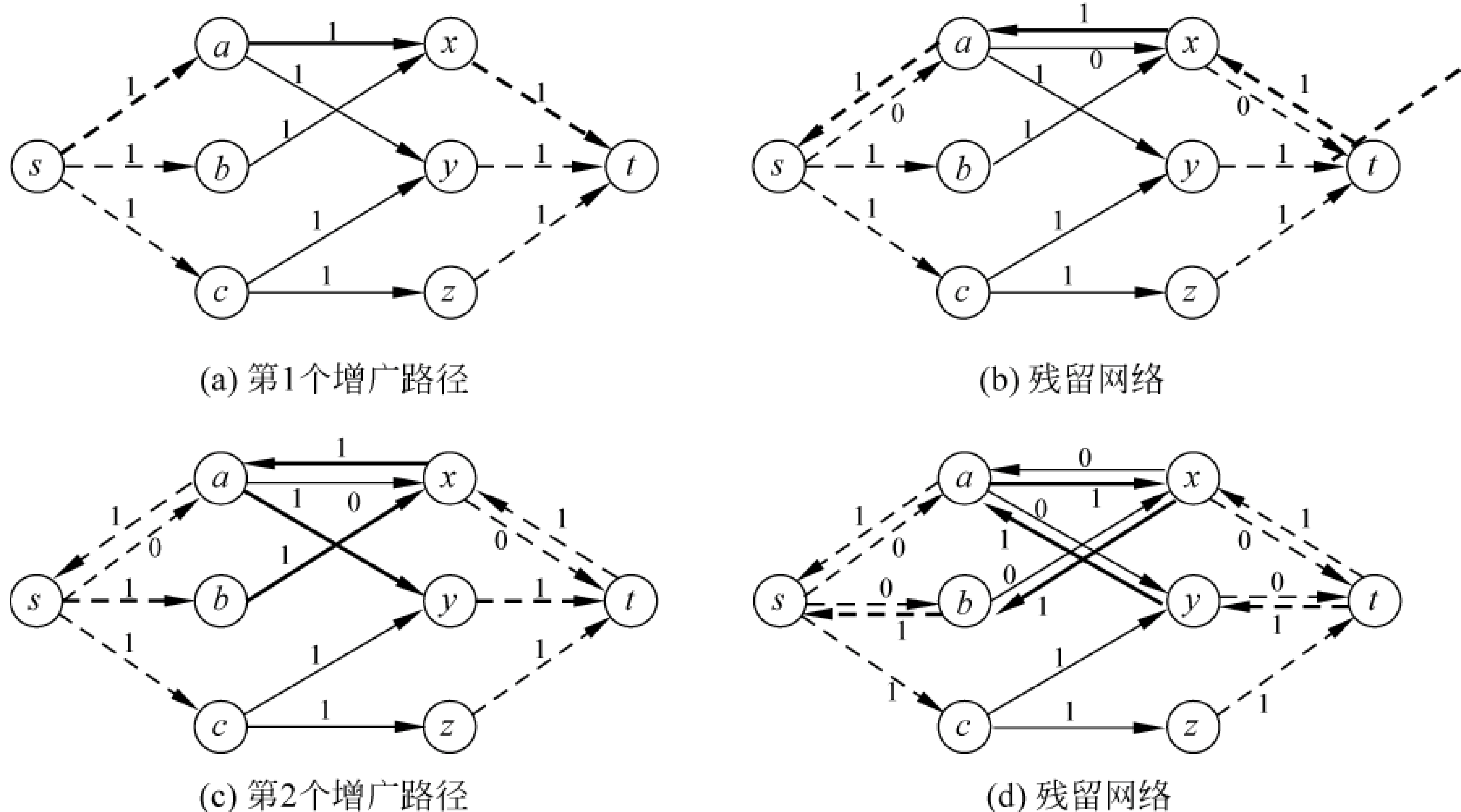


图10.34 用最大流来求解

(1) 找到第1个增广路径，找到匹配a-x，见图10.34(a)。

(2) 更新残留网络，见图10.34(b)。

(3) 找到第2个增广路径，找到匹配a-y、b-x。在这一步把原来的配对a-x改为a-y，以成全b-x的配对，这就是残留网络的作用，见图10.34(c)。

(4) 更新残留网络，见图10.34(d)。

后面的步骤请读者做练习。

3. 匈牙利算法

匈牙利算法可以看成最大流的一个特殊实现。

由于二分图是一个很简单的图，并不需要按上面的图解做标准的最大流，可以进行简化。

(1) 从上面的图解中发现对s和t的操作是多余的，直接从a、b、c开始找增广路径就可以了。

(2) 残留网络上的增广路需要覆盖完整的路径,如果在二分图中只进行$\{a,b,c\}$到$\{x,y,z\}$的局部操作,将简化很多。

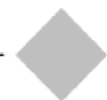
下面是 hdu 2063 的程序。

hdu 2063 匈牙利算法(邻接矩阵)

```
#include<bits/stdc++.h>
using namespace std;
int G[510][510];
int match[510], reserve_boy[510];            //匹配结果在 match[]中
int k, m_girl, n_boy;
bool dfs(int x){                             //找一个增广路径,即给女孩 x 找一个配对男孩
    for(int i=1; i<=n_boy; i++)
        if(!reserve_boy[i] && G[x][i]){
            reserve_boy[i] = 1;              //预定男孩 i,准备分给女孩 x
            if(!match[i] || dfs(match[i])){
    //有两种情况:(1)如果男孩 i 还没配对,就分给女孩 x;
    //(2)如果男孩 i 已经配对,尝试用 dfs()更换原配女孩,以腾出位置给女孩 x
                match[i] = x;
                        //配对成功.如果原来有配对,更换成功.现在男孩 i 属于女孩 x
                return true;
            }
        }
    return false;                            //女孩 x 没有喜欢的男孩,或者更换不成功
}
int main(){
    while(scanf("%d",&k)!= EOF && k){
        scanf("%d%d",&m_girl,&n_boy);
        memset(G,0,sizeof(G));
        memset(match,0,sizeof(match));
        for(int i=0;i<k;i++){
            int a,b;
            scanf("%d%d",&a,&b);
            G[a][b]=1;
        }
        int sum=0;
        for(int i=1; i<=m_girl; i++){        //为每个女孩找配对
            memset(reserve_boy,0,sizeof(reserve_boy));
            if(dfs(i))  sum++;
                //第 i 个女孩配对成功,这个配对后面可能会更换,但是保证她能配对
        }
        printf("%d\n",sum);
    }
    return 0;
}
```

上述程序用邻接矩阵,找一次增广路径的时间复杂度为 $O(V^2)$,总时间为 $O(V^3)$;空间复杂度为 $O(V^2)$。

改用邻接表存图可以加快搜索速度。找一次增广路径的时间复杂度为 $O(V+E)$,总时



间为 $O(VE)$；空间复杂度为 $O(V+E)$。读者可以练习把上述程序改成用邻接表。

【习题】

hdu 1083 “Courses”，简单题。

hdu 3729 “I'm Telling the Truth”，简单题。

hdu 5727 “Necklace”。

hdu 3605 “Escape”，二分图多重匹配。

10.15　小　　结

本章讲解了很多图论问题，关键的知识点总结如下。

(1) 图的存储：牢固掌握图的邻接矩阵、邻接表、链式前向星 3 种存储方法。

(2) BFS 和 DFS 在图问题中的关键作用：DFS 对图的遍历过程。

(3) 拓扑排序：BFS 和 DFS 的直接应用。

(4) 欧拉路：DFS 的直接应用。

(5) 无向图连通性：缩点的方法。

(6) 有向图连通性：DFS 的深度应用。

(7) 2-SAT 问题：强连通分量和拓扑排序的应用。

(8) 最短路径：透彻掌握各种最短路径算法的思想、数据结构、编程、应用环境。

(9) 最小生成树：贪心法思想的应用。

(10) 最大流：网络流的基础问题；残留网络、增广路的方法。请读者透彻掌握。

(11) 最小割：问题建模。

(12) 最小费用最大流：最短路径和最大流的结合。

(13) 二分图匹配：最大流思想的应用。

第 11 章　计算几何

✍ 二维几何基础

✍ 圆

✍ 三维几何

✍ 几何模板

几何类题目是算法竞赛中的一个大类考点，涉及的知识点有平面几何、解析几何、计算几何等。

几何题的代码一般比较长，有时甚至有 200 多行，而且逻辑往往也比较复杂，是典型的考查参赛人员编码能力的题型。

如果要选出带到赛场的必备模板，其中一定会包括几何模板。很多有经验的老队员说："做几何题，模板很重要，要高度可靠！"因此，在平时的训练过程中认真总结模板，融会贯通，才能在赛场上灵活地使用它们。

11.1　二维几何基础

计算几何中的坐标值一般是实数，在编程时用 double 类型，不用精度较低的 float 类型。double 类型读入时用%lf 格式，输出时用%f 格式。



视频讲解

在进行浮点数运算时会产生精度误差，为了控制精度，可以设置一个偏差值 eps(epsilon)，eps 要大于浮点运算结果的不确定量，一般取 10^{-8}。如果 eps 取 10^{-10}，可能会有问题，例如 11.2.1 节中提到的 hdu 5572 题，用 10^{-10} 会返回 Wrong Answer。

判断一个浮点数是否等于 0，不能直接用"==0"来判断，而是用 sgn()函数判断是否小于 eps。在比较两个浮点数时，也不能直接用"=="判断是否相等，而是用 dcmp()函数判断是否相等。

```
const double pi = acos(-1.0);           //高精度圆周率
const double eps = 1e-8;                //偏差值，有时用 1e-10
int sgn(double x){                      //判断 x 是否等于 0
    if(fabs(x) < eps)  return 0;
    else return x<0?-1:1;
}
int dcmp(double x, double y){           //比较两个浮点数：0 为相等；-1 为小于；1 为大于
    if(fabs(x - y) < eps) return 0;
    else return x<y ?-1:1;
}
```



11.1.1 点和向量

1. 点

二维平面中的点用坐标(x, y)来表示。

```
struct Point{
    double x,y;
    Point(){}
    Point(double x,double y):x(x),y(y){}
};
```

2. 两点之间的距离

(1) 把两点看成直角三角形的两个顶点,斜边就是两点的距离。用库函数 hypot()计算直角三角形的斜边长。

```
double Distance(Point A, Point B){return hypot(A.x - B.x,A.y - B.y);}
```

(2) 或者用 sqrt()函数计算。

```
double Dist(Point A,Point B){
    return sqrt((A.x - B.x) * (A.x - B.x)  +  (A.y - B.y) * (A.y - B.y));
}
```

3. 向量

有大小、有方向的量称为向量(矢量),只有大小没有方向的量称为标量。

用平面上的两个点可以确定一个向量,例如用起点 P_1 和终点 P_2 表示一个向量。不过,为了简化描述,可以把它平移到原点,把向量看成从原点(0,0)指向点(x,y)的一个有向线段,如图 11.1 所示。向量的表示在形式上与点的表示完全一样,可以用点的数据结构来表示向量:

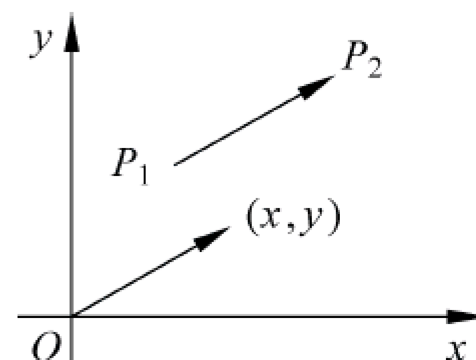


图 11.1 向量

```
typedef Point Vector;
```

注意,向量并不是一个有向线段,只是表示方向和大小,所以向量平移后仍然不变。



视频讲解

4. 向量的运算

在 struct Point 中,对向量运算重载运算符。

(1) 加:点与点的加法运算没有意义;点与向量相加得到另一个点;向量与向量相加得到另外一个向量。

```
Point operator + (Point B){return Point(x + B.x,y + B.y);}
```

(2) 减:两个点的差是一个向量;向量 $\boldsymbol{A}$ 减 $\boldsymbol{B}$ 得到由 $\boldsymbol{B}$ 指向 $\boldsymbol{A}$ 的向量。

```
Point operator  -  (Point B){return Point(x - B.x,y - B.y);}
```

向量的加法和减法示意图如图 11.2 所示。



图 11.2　向量的加法和减法

(3) 乘：向量与实数相乘得到等比例放大的向量。

```
Point operator * (double k){return Point(x*k,y*k);}
```

(4) 除：向量与实数相除得到等比例缩小的向量。

```
Point operator / (double k){return Point(x/k,y/k);}
```

(5) 等于：

```
bool operator == (Point B){return sgn(x - B.x) == 0 && sgn(y - B.y) == 0;}
```

11.1.2　点积和叉积

向量的基本运算是点积和叉积，计算几何的各种操作几乎都基于这两种运算。

1. 点积（Dot product）

记向量 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 的点积为 $\boldsymbol{A}\cdot\boldsymbol{B}$，定义如下：

$$\boldsymbol{A}\cdot\boldsymbol{B}=|\boldsymbol{A}||\boldsymbol{B}|\cos\theta$$

其中 θ 为 $\boldsymbol{A}$、$\boldsymbol{B}$ 之间的夹角。点积的几何意义为 $\boldsymbol{A}$ 在 $\boldsymbol{B}$ 上的投影长度乘以 $\boldsymbol{B}$ 的模长。点积的几何表示如图 11.3 所示。

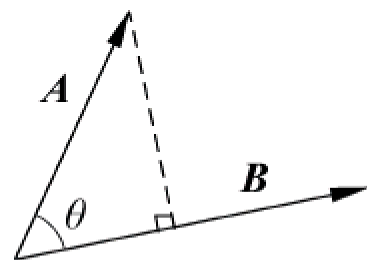


图 11.3　点积的几何表示

在编程时计算点积并不需要知道 θ。如果已知 $\boldsymbol{A}=(\boldsymbol{A}.\mathrm{x},\boldsymbol{A}.\mathrm{y})$，$\boldsymbol{B}=(\boldsymbol{B}.\mathrm{x},\boldsymbol{B}.\mathrm{y})$，那么有：

$$\boldsymbol{A}\cdot\boldsymbol{B}=\boldsymbol{A}.\mathrm{x}*\boldsymbol{B}.\mathrm{x}+\boldsymbol{A}.\mathrm{y}*\boldsymbol{B}.\mathrm{y}$$

下面推导这个公式。设 θ1 是 $\boldsymbol{A}$ 与 x 轴的夹角，θ2 是 $\boldsymbol{B}$ 与 x 轴的夹角，向量 $\boldsymbol{A}$ 与 $\boldsymbol{B}$ 的夹角 θ 等于 θ1−θ2，那么有：

$$\begin{aligned}&\boldsymbol{A}.\mathrm{x}*\boldsymbol{B}.\mathrm{x}+\boldsymbol{A}.\mathrm{y}*\boldsymbol{B}.\mathrm{y}\\&=(|\boldsymbol{A}|*\cos\theta1)*(|\boldsymbol{B}|*\cos\theta2)+(|\boldsymbol{A}|*\sin\theta1)*(|\boldsymbol{B}|*\sin\theta2)\\&=|\boldsymbol{A}||\boldsymbol{B}|(\cos\theta1*\cos\theta2+\sin\theta1*\sin\theta2)\\&=|\boldsymbol{A}||\boldsymbol{B}|(\cos(\theta1-\theta2))\\&=|\boldsymbol{A}||\boldsymbol{B}|\cos\theta\end{aligned}$$

求 $\boldsymbol{A}$、$\boldsymbol{B}$ 点积的代码如下：

```
double Dot(Vector A,Vector B){return A.x*B.x + A.y*B.y;}
```

2. 点积的应用

1) 判断 $\boldsymbol{A}$ 与 $\boldsymbol{B}$ 的夹角是钝角还是锐角

点积有正负，利用正负号可以判断向量的夹角：

若 $\mathrm{dot}(\boldsymbol{A},\boldsymbol{B})>0$，$\boldsymbol{A}$ 与 $\boldsymbol{B}$ 的夹角为锐角；

若 dot($\boldsymbol{A}$,$\boldsymbol{B}$)<0,$\boldsymbol{A}$ 与 $\boldsymbol{B}$ 的夹角为钝角;
若 dot($\boldsymbol{A}$,$\boldsymbol{B}$)=0,$\boldsymbol{A}$ 与 $\boldsymbol{B}$ 的夹角为直角。
2) 求向量 $\boldsymbol{A}$ 的长度

```
double Len(Vector A){return sqrt(Dot(A,A));}
```

或者是求长度的平方,避免开方运算:

```
double Len2(Vector A){return Dot(A,A);}
```

3) 求向量 $\boldsymbol{A}$ 与 $\boldsymbol{B}$ 的夹角大小

```
double Angle(Vector A,Vector B){return acos(Dot(A,B)/Len(A)/Len(B));}
```

3. 叉积(Cross product)

叉积是比点积更常用的几何概念。它的计算公式如下:

$$\boldsymbol{A}\times\boldsymbol{B}=|\boldsymbol{A}||\boldsymbol{B}|\sin\theta$$

θ 表示向量 $\boldsymbol{A}$ 旋转到向量 $\boldsymbol{B}$ 所经过的夹角。

两个向量的叉积是一个带正负号的数值。$\boldsymbol{A}\times\boldsymbol{B}$ 的几何意义为向量 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 形成的平行四边形的"有向"面积,这个面积是有正负的。叉积的正负符合"右手定则",读者可以用图 11.4 中的正负情况帮助理解。

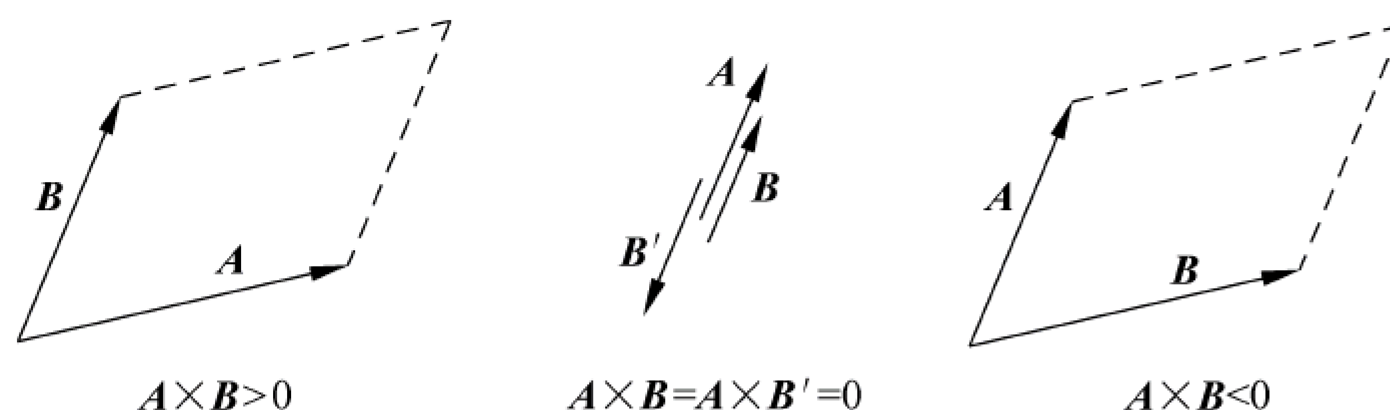


图 11.4　叉积与叉积的正负

用以下程序计算向量 $\boldsymbol{A}$、$\boldsymbol{B}$ 的叉积 $\boldsymbol{A}\times\boldsymbol{B}$:

```
double Cross(Vector A,Vector B){return A.x*B.y - A.y*B.x;}
```

对于其正确性,读者可以用前文证明点积的推导方法来证明。
注意函数 Cross()中的 $\boldsymbol{A}$、$\boldsymbol{B}$ 是有顺序的,叉积有正负,$\boldsymbol{A}\times\boldsymbol{B}$ 与 $\boldsymbol{B}\times\boldsymbol{A}$ 相反。
叉积有正负,这个性质使得叉积能用于很多有用的场合。

4. 叉积的基本应用

下面给出叉积的几个基本应用。对于其他应用,例如求两个线段的方向关系、求多边形的面积等,将在后文讲解。

1) 判断向量 $\boldsymbol{A}$、$\boldsymbol{B}$ 的方向关系
若 $\boldsymbol{A}\times\boldsymbol{B}>0$,$\boldsymbol{B}$ 在 $\boldsymbol{A}$ 的逆时针方向;
若 $\boldsymbol{A}\times\boldsymbol{B}<0$,$\boldsymbol{B}$ 在 $\boldsymbol{A}$ 的顺时针方向;
若 $\boldsymbol{A}\times\boldsymbol{B}=0$,$\boldsymbol{B}$ 与 $\boldsymbol{A}$ 共线,可能是同方向的,也可能是反方向的。

2）计算两向量构成的平行四边形的有向面积

3 个点 $\boldsymbol{A}$、$\boldsymbol{B}$、$\boldsymbol{C}$，以 $\boldsymbol{A}$ 为公共点，得到两个向量 $\boldsymbol{B}-\boldsymbol{A}$ 和 $\boldsymbol{C}-\boldsymbol{A}$，它们构成的平行四边形的面积如下：

```
double Area2(Point A,Point B,Point C){return Cross(B-A, C-A);}
```

如果以 $\boldsymbol{B}$ 或 $\boldsymbol{C}$ 为公共点构成平行四边形，面积是相等的，但是正负不一样。

3）计算 3 点构成的三角形的面积

3 个点 $\boldsymbol{A}$、$\boldsymbol{B}$、$\boldsymbol{C}$ 构成的三角形的面积等于平行四边形面积 Area2($\boldsymbol{A}$,$\boldsymbol{B}$,$\boldsymbol{C}$)的 1/2。

4）向量旋转

使向量(x,y)绕起点逆时针旋转，设旋转角度为 θ，那么旋转后的向量(x',y')如下：

$$x' = x\cos\theta - y\sin\theta$$
$$y' = x\sin\theta + y\cos\theta$$

代码如下，向量 $\boldsymbol{A}$ 逆时针旋转的角度为 rad：

```
Vector Rotate(Vector A, double rad){
    return Vector(A.x*cos(rad) - A.y*sin(rad), A.x*sin(rad) + A.y*cos(rad));
}
```

特殊情况是旋转 90°。

逆时针旋转 90°：Rotate(A, pi/2)，返回 Vector($-A.y$, $A.x$)；

顺时针旋转 90°：Rotate(A, $-$pi/2)，返回 Vector($A.y$,$-A.x$)。

有时需要求单位法向量，即逆时针转 90°，然后取单位值。代码如下：

```
Vector Normal(Vector A){return Vector(-A.y/Len(A), A.x/Len(A));}
```

5）用叉积检查两个向量是否平行或重合

```
bool Parallel(Vector A, Vector B){return sgn(Cross(A,B)) == 0;}
```

11.1.3 点和线

1. 直线的表示

直线有多种表示方法，用户在编程时可以灵活使用这些方法：

(1) 用直线上的两个点来表示。

(2) $ax+by+c=0$，普通式。

(3) $y=kx+b$，斜截式。

(4) $P=P_0+vt$，点向式。也就是用 P_0 和 v 来表示直线 P，t 是变量，可以取任意值。$P_0(x_0,y_0)$是直线上的一个点；v 是方向向量，给定两个点 A、B，那么 $v=B-A$。

点向式非常便于计算机处理，也能方便地表示射线、线段：

如果 t 无限制，P 是直线；

如果 t 在$[0,1]$内，P 是 A、B 之间的线段；

如果 $t>0$，P 是射线。

```
struct Line{
```

```
    Point p1,p2;                        //线上的两个点
    Line(){}
    Line(Point p1,Point p2):p1(p1),p2(p2){}
    //根据一个点和倾斜角 angle 确定直线,0<=angle<pi
    Line(Point p,double angle){
        p1 = p;
        if(sgn(angle - pi/2) == 0){p2 = (p1 + Point(0,1));}
        else{p2 = (p1 + Point(1,tan(angle)));}
    }
    //ax + by + c = 0
    Line(double a,double b,double c){
        if(sgn(a) == 0){
            p1 = Point(0, -c/b);
            p2 = Point(1, -c/b);
        }
        else if(sgn(b) == 0){
            p1 = Point(-c/a,0);
            p2 = Point(-c/a,1);
        }
        else{
            p1 = Point(0, -c/b);
            p2 = Point(1,(-c-a)/b);
        }
    }
};
```

2. 线段的表示

可以用两个点表示线段,起点是 p_1,终点是 p_2。直接用直线的数据结构定义线段:

```
typedef Line Segment;
```

3. 点和直线的位置关系

在二维平面上,点和直线有 3 种位置关系,即点在直线左侧、在右侧、在直线上。用直线上的两点 p_1 和 p_2 与点 p 构成两个向量,用叉积的正负判断方向,就能得到位置关系。

```
int Point_line_relation(Point p, Line v){
    int c = sgn(Cross(p-v.p1,v.p2-v.p1));
    if(c < 0)return 1;                  //1: p 在 v 的左边
    if(c > 0)return 2;                  //2: p 在 v 的右边
    return 0;                           //0: p 在 v 上
}
```

4. 点和线段的位置关系

判断点 p 是否在线段 v 上，先用叉积判断是否共线，然后用点积看 p 和 v 的两个端点产生的角是否为钝角(实际上应该是 180°角)。

```
bool Point_on_seg(Point p, Line v){         //点和线段: 0 为点不在线段 v 上; 1 为点在线段 v 上
    return sgn(Cross(p-v.p1, v.p2-v.p1)) == 0 && sgn(Dot(p - v.p1,p- v.p2)) <= 0;
}
```

5. 点到直线的距离

已知点 p 和直线 $v(p_1,p_2)$，求 p 到 v 的距离。首先用叉积求 p、p_1、p_2 构成的平行四边形的面积，然后用面积除以平行四边形的底边长，也就是线段(p_1, p_2)的长度，就得到了平行四边形的高，即 p 点到直线的距离。

```
double Dis_point_line(Point p, Line v){
    return fabs(Cross(p - v.p1,v.p2 - v.p1))/Distance(v.p1,v.p2);
}
```

6. 点在直线上的投影

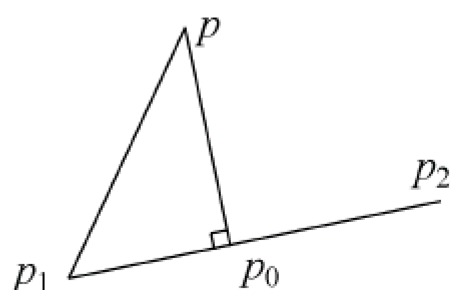


图 11.5 点在直线上的投影

已知直线上的两点 p_1、p_2 以及直线外的一点 p，求投影点 p_0，如图 11.5 所示。

令 $k=\frac{|p_0-p_1|}{|p_2-p_1|}$，即 k 是线段 $p_0\,p_1$ 和 $p_2\,p_1$ 长度的比值。因为 $p_0=p_1+k*(p_2-p_1)$，如果求得 k，就能得到 p_0。

根据点积的概念，有

$$(p-p_1)\cdot(p_2-p_1)=|p_2-p_1|*|p_0-p_1|$$

即 $|p_0-p_1|=\frac{(p-p_1)\cdot(p_2-p_1)}{|p_2-p_1|}$，代入得

$$k=\frac{|p_0-p_1|}{|p_2-p_1|}=\frac{(p-p_1)\cdot(p_2-p_1)}{|p_2-p_1|*|p_2-p_1|}$$

所以

$$p_0=p_1+k*(p_2-p_1)=p_1+\frac{(p-p_1)\cdot(p_2-p_1)}{|p_2-p_1|*|p_2-p_1|}*(p_2-p_1)$$

代码如下：

```
Point Point_line_proj(Point p, Line v){
    double k = Dot(v.p2 - v.p1,p - v.p1)/Len2(v.p2 - v.p1);
    return v.p1 + (v.p2 - v.p1) * k;
}
```

7. 点关于直线的对称点

求一个点 p 对一条直线 v 的镜像点。先求点 p 在直线上的投影 q，再求对称点 p'，如图 11.6 所示。

```
Point Point_line_symmetry(Point p, Line v){
    Point q = Point_line_proj(p,v);
    return Point(2 * q.x - p.x,2 * q.y - p.y);
}
```

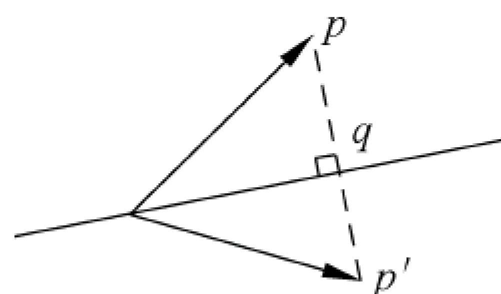


图 11.6 对称点

8. 点到线段的距离

对于点 p 到线段 AB 的距离，在以下 3 个距离中取最小值：从 p 出发对 AB 做垂线，如果交点在 AB 线段上，这个距离就是最小值；p 到 A 的距离；p 到 B 的距离。

```
double Dis_point_seg(Point p, Segment v){
```

```
    if(sgn(Dot(p - v.p1,v.p2 - v.p1))< 0 || sgn(Dot(p - v.p2,v.p1 - v.p2))< 0)
        return min(Distance(p,v.p1),Distance(p,v.p2));
    return Dis_point_line(p,v);            //点的投影在线段上
}
```

9. 两条直线的位置关系

```
int Line_relation(Line v1, Line v2){
    if(sgn(Cross(v1.p2 - v1.p1,v2.p2 - v2.p1))  ==  0){
        if(Point_line_relation(v1.p1,v2) == 0) return 1;       //1:重合
        else return 0;                                         //0:平行
    }
    return 2;                                                  //2:相交
}
```

10. 求两条直线的交点

对于两直线的交点,可以通过 $a_1x+b_1y+c_1=0$ 与 $a_2x+b_2y+c_2=0$ 联立方程求解。不过,借助叉积有更简单的方法。

图 11.7 中有 4 个点 A、B、C、D,组成两条直线 AB 和 CD,交点是 P。以下两个关系成立:

$\frac{|DP|}{|CP|}=\frac{S_{\triangle ABD}}{S_{\triangle ABC}}=\frac{\overrightarrow{AD}\times\overrightarrow{AB}}{\overrightarrow{AB}\times\overrightarrow{AC}}$,其中 $S_{\triangle ABD}$、$S_{\triangle ABC}$ 表示三角形的面积。

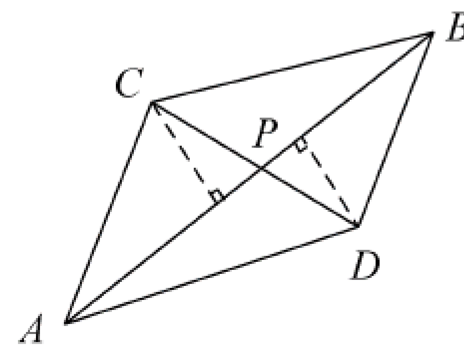


图 11.7 直线的交点

$\frac{|DP|}{|CP|}=\frac{x_D-x_P}{x_P-x_C}=\frac{y_D-y_P}{y_P-y_C}$,其中 x_D、y_D 等表示点的坐标。

联立上面两个方程,得到交点 P 的坐标如下:

$$x_P=\frac{S_{\triangle ABD}\times x_C+S_{\triangle ABC}\times x_D}{S_{\triangle ABD}+S_{\triangle ABC}}$$

$$y_P=\frac{S_{\triangle ABD}\times y_C+S_{\triangle ABC}\times y_D}{S_{\triangle ABD}+S_{\triangle ABC}}$$

三角形的面积可以通过叉积求得:$S_{\triangle ABD}=\overrightarrow{AD}\times\overrightarrow{AB}$,$S_{\triangle ABC}=\overrightarrow{AB}\times\overrightarrow{AC}$。

程序如下:

```
Point Cross_point(Point a,Point b,Point c,Point d){      //Line1:ab, Line2:cd
   double s1  =  Cross(b - a,c - a);
   double s2  =  Cross(b - a,d - a);                     //叉积有正负
   return Point(c.x * s2 - d.x * s1,c.y * s2 - d.y * s1)/(s2 - s1);
}
```

注意:在 Cross_point()中要对(s2−s1)做除法,所以在调用 Cross_point()之前应该保证 s2−s1≠0,即直线 AB、CD 不共线,而且不平行。

11. 判断两个线段是否相交

这里仍然利用叉积有正负的特点。如果一条线段的两端在另一条线段的两侧,那么两个端点与另一线段产生的两个叉积正负相反,也就是说两个叉积相乘为负。如果两条线段互相满足这一点,那么就是相交的。

```
bool Cross_segment(Point a,Point b,Point c,Point d){    //Line1:ab;   Line2:cd
    double c1 = Cross(b - a,c - a),c2 = Cross(b - a,d - a);
    double d1 = Cross(d - c,a - c),d2 = Cross(d - c,b - c);
    return sgn(c1) * sgn(c2)< 0 && sgn(d1) * sgn(d2)< 0;   //1:相交; 0:不相交
}
```

12. 求两条线段的交点

先判断两条线段是否相交，若相交，问题转化成两条直线求交点。

11.1.4 多边形

1. 判断点在多边形内部

给定一个点 P 和一个多边形，判断 P 是否在多边形内部，有射线法和转角法两种方法。

射线法：从 P 引一条射线，穿过多边形，如果和多边形的边相交奇数次，说明 P 在外部；如果是偶数次，说明在内部。这种方法比较烦琐，很少使用。

转角法：把点 P 和多边形的每个点连接，逐个计算角度，绕多边形一周，看多边形相对于这个点总共转了多少度。如果是 360°，说明点在多边形内；如果是 0°，说明点在多边形外；如果是 180°，说明点在多边形边界上。但是，如果直接算角度，需要计算反三角函数，不仅速度慢，而且有精度问题。

下面的方法是转角法思想的另一种实现：以点 P 为起点引一条水平线，检查与多边形每条边的相交情况，例如沿着逆时针，检查 P 和每个边的相交情况，统计 P 穿过这些边的次数。见图 11.8 和图 11.9，检查以下 3 个参数：

$$c = \text{Cross}(P - j, i - j)$$
$$u = i.y - P.y$$
$$v = j.y - P.y$$

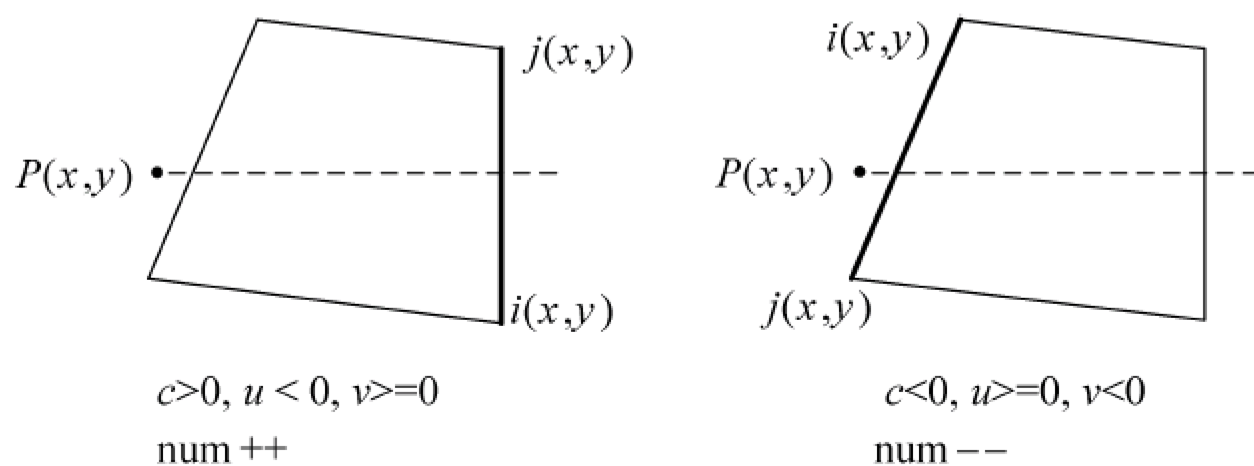


图 11.8　P 在多边形左侧

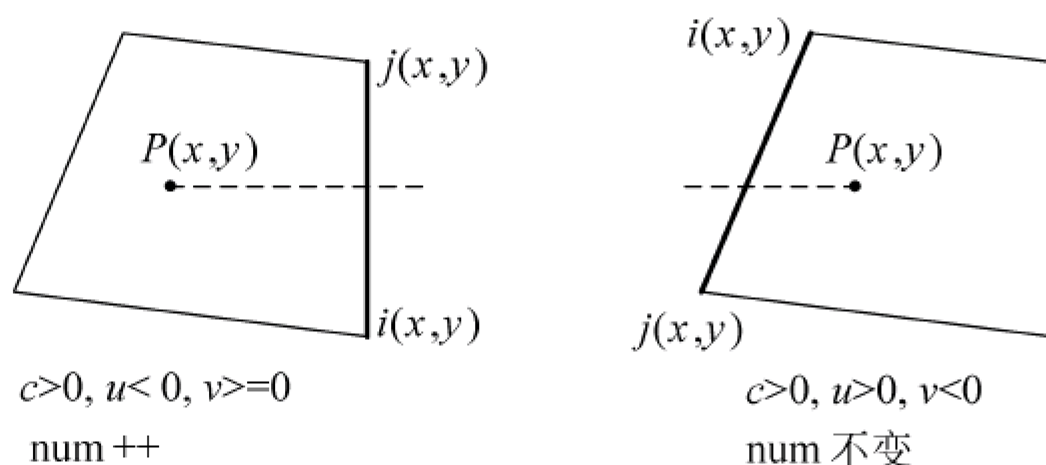


图 11.9　P 在多边形内部

叉积 c 用来检查 P 点在线段 ij 的左侧还是右侧，u、v 用来检查经过 P 的水平线是否穿过线段 ij。

用 num 计数：

```
if(c > 0 && u < 0 && v >= 0) num++;
if(c < 0 && u >= 0 && v < 0) num--;
```

当 num>0 时，P 在多边形内部。读者可以验证其他情况，例如 P 在多边形右侧、多边形是凹多边形，看上述判断是否成立。

下面是代码，注意多边形的形状是由各个顶点的排列顺序决定的。

```
int Point_in_polygon(Point pt,Point *p,int n){          //点 pt,多边形 Point *p
    for(int i = 0;i < n;i++){                           //3: 点在多边形的顶点上
        if(p[i] == pt)return 3;
    }
    for(int i = 0;i < n;i++){                           //2: 点在多边形的边上
        Line v = Line(p[i],p[(i+1)%n]);
        if(Point_on_seg(pt,v)) return 2;
    }
    int num = 0;
    for(int i = 0;i < n;i++){
        int j = (i+1) % n;
        int c = sgn(Cross(pt - p[j],p[i] - p[j]));
        int u = sgn(p[i].y - pt.y);
        int v = sgn(p[j].y - pt.y);
        if(c > 0 && u < 0 && v >= 0) num++;
        if(c < 0 && u >= 0 && v < 0) num--;
    }
    return num != 0;                                    //1: 点在内部; 0: 点在外部
}
```

2. 求多边形的面积

给定一个凸多边形，求它的面积。读者很容易想到，可以在凸多边形内部找一个点 P，然后以这个点为中心，与凸多边形的边结合，对多边形进行三角剖分，所有三角形的和就是凸多边形的面积。每个三角形的面积可以用叉积来求。

事实上，上述方法不仅可用于凸多边形，也适用于非凸多边形；而且点 P 并不需要在多边形内部，在任何位置都可以，例如以原点为 P，编程最简单。这是因为叉积是有正负的，它可以抵消多边形外部的面积，图 11.10 给出了各种情况。

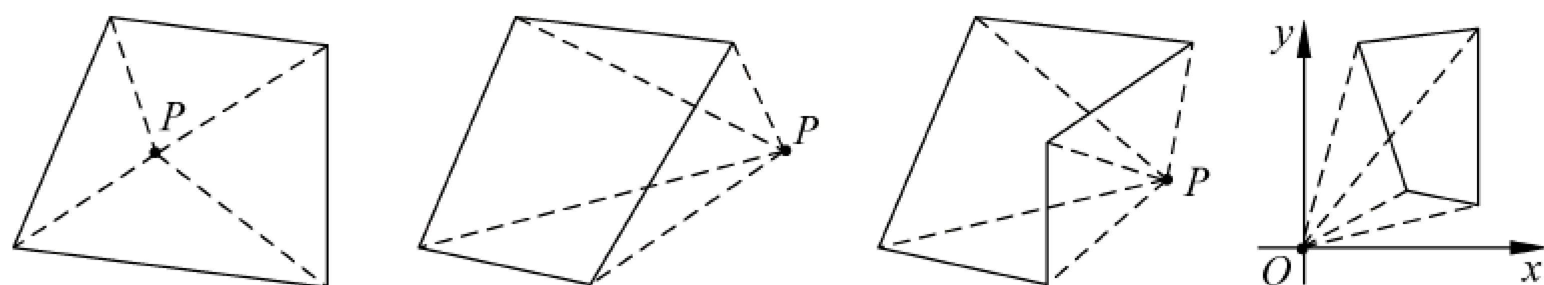


图 11.10 求任意多边形的面积

下面的程序以原点为中心点划分三角形，然后求多边形的面积。

```
double Polygon_area(Point *p, int n){                   //Point *p 表示多边形
    double area = 0;
```

```
    for(int i = 0;i < n;i++)
        area += Cross(p[i],p[(i + 1) % n]);
    return area/2;                          //面积有正负,这里不能简单地取绝对值
}
```

3. 求多边形的重心

将多边形三角剖分,算出每个三角形的重心,三角形的重心是 3 点坐标的平均值,然后对每个三角形的有向面积求加权平均。

下面用一个例题综合讲解前面一些模板的应用。代码中的 Polygon_center()是求多边形的重心。

> **hdu 1115 "Lifting the Stone"**
>
> 给定一个 N 多边形,$3 \leqslant N \leqslant 1\,000\,000$,求重心。

代码如下:

```
#include <bits/stdc++.h>
struct Point{
    double x,y;
    Point(double X = 0,double Y = 0){x = X,y = Y;}
    Point operator + (Point B){return Point (x + B.x,y + B.y);}
    Point operator - (Point B){return Point (x - B.x,y - B.y);}
    Point operator * (double k){return Point (x * k,y * k);}
    Point operator / (double k){return Point (x/k,y/k);}
};
typedef Point Vector;
double Cross(Vector A,Vector B){return A.x * B.y - A.y * B.x;}
double Polygon_area(Point *p, int n){             //求多边形的面积
    double area = 0;
    for(int i = 0;i < n;i++)
        area += Cross(p[i],p[(i + 1) % n]);
    return area/2;                                //面积有正负,不能取绝对值
}
Point Polygon_center(Point *p, int n){            //求多边形的重心
    Point ans(0,0);
    if(Polygon_area(p,n) == 0) return ans;
    for(int i = 0;i < n;i++)
        ans = ans + (p[i] + p[(i + 1) % n]) * Cross(p[i],p[(i + 1) % n]);
    return ans/Polygon_area(p,n)/6;
}
int main(){
    int t,n,i;
    Point center;                                 //重心的坐标
    Point p[100000];
    scanf("%d",&t);
    while(t--){
        scanf("%d",&n);
        for(i = 0;i < n;i++) scanf("%lf %lf",&p[i].x,&p[i].y);
```

```
        center = Polygon_center(p,n);
        printf("%.2f %.2f\n",center.x,center.y);  //注意这里输出用%f,不是用%lf
    }
    return 0;
}
```

【习题】

hdu 1558,几何+并查集。

11.1.5 凸包

凸包(Convex hull)是计算几何中的著名问题,有非常广泛的应用[①]。

凸包问题:给定一些点,求能把所有这些点包含在内的面积最小的多边形。可以想象有一个很大的橡皮箍,它把所有的点都箍在里面,在橡皮箍收紧之后,绕着最外围的点形成的多边形就是凸包。

求凸包的常用算法有两种,一是 Graham 扫描法,其复杂度是 $O(n\log_2 n)$;二是 Jarvis 步进法,其复杂度是 $O(nh)$,h 是凸包上的顶点数。这两种算法的基本思路是"旋转扫除",设定一个参照顶点,逐个旋转到其他所有顶点,并判断这些顶点是否在凸包上。

这里介绍 Graham 扫描法的变种——Andrew 算法,它更快、更稳定。算法做两次扫描,先从最左边的点沿"下凸包"扫描到最右边,再从最右边的点沿"上凸包"扫描到最左边,"上凸包"和"下凸包"合起来就是完整的凸包。

具体步骤如下:

(1) 把所有点按照横坐标 x 从小到大进行排序,如果 x 相同,按 y 从小到大排序,并删除重复的点,得到序列$\{p_0, p_1, p_2, \cdots, p_m\}$。

(2) 从左到右扫描所有点,求"下凸包"。p_0 一定在凸包上,它是凸包最左边的顶点,从 p_0 开始,依次检查$\{p_1, p_2, \cdots, p_m\}$,扩展出"下凸包"。判断的依据是:如果新点在凸包"前进"方向的左边,说明在"下凸包"上,把它加入到凸包;如果在右边,说明拐弯了,删除最近加入下凸包的点。继续这个过程,直到检查完所有点。拐弯方向用叉积判断即可。例如图 11.11 所示,在检查 p_4 时发现 p_4p_3 对 p_3p_2 是右拐弯的,说明 p_3 不在下凸包上(有可能在"上凸包"上,在步骤(3)中会判断);退回到 p_2,发现 p_4p_2 对 p_2p_1 也是右拐弯的,退回到 p_1。

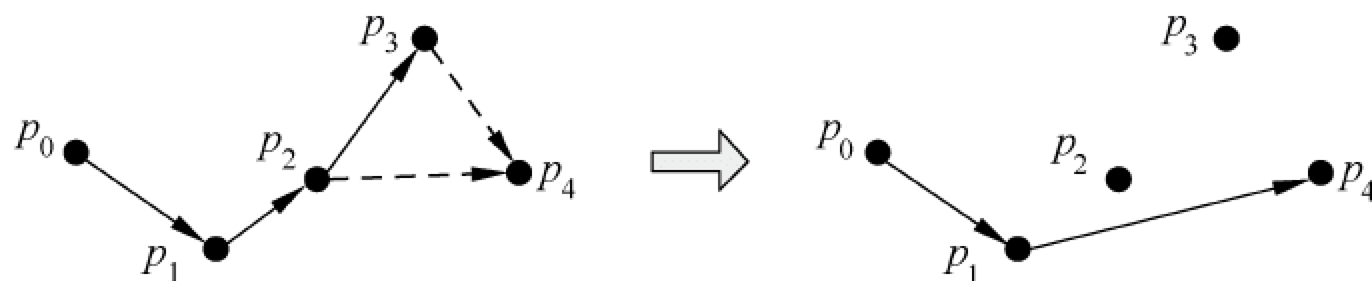


图 11.11 下凸包

① https://en.wikipedia.org/wiki/Convex_hull。

(3) 从右到左重新扫描所有点，求“上凸包”。和求“下凸包”的过程类似，最右边的点 p_m 一定在凸包上。

复杂度。算法先对点排序，复杂度是 $O(n\log_2 n)$[①]，然后扫描 $O(n)$ 次得到凸包。算法的总复杂度是 $O(n\log_2 n)$。

下面用一个例题讲解凸包模板的应用。代码中的 Convex_hull()是求凸包，注意其中用于去重的 unique()函数。

hdu 1392 “Surround the Trees”

输入 n 个点，求凸包的周长。

代码如下：

```
#include<bits/stdc++.h>
using namespace std;
const int maxn = 104;
const double eps = 1e-8;
int sgn(double x){                    //判断 x 是否等于 0
    if(fabs(x) < eps)  return 0;
    else return x<0?-1:1;
}
struct Point{
    double x,y;
    Point(){}
    Point(double x, double y):x(x),y(y){}
    Point operator + (Point B){return Point(x+B.x,y+B.y);}
    Point operator - (Point B){return Point(x-B.x,y-B.y);}
    bool operator == (Point B){return sgn(x-B.x) == 0 && sgn(y-B.y) == 0;}
    bool operator <(Point B){         //用于 sort()排序
        return sgn(x-B.x)<0 || (sgn(x-B.x)==0 && sgn(y-B.y)<0);}
};
typedef Point Vector;
double Cross(Vector A,Vector B){return A.x*B.y - A.y*B.x;} //叉积
double Distance(Point A,Point B){return hypot(A.x-B.x,A.y-B.y);}
//Convex_hull()求凸包.凸包顶点放在 ch 中,返回值是凸包的顶点数
int Convex_hull(Point *p,int n,Point *ch){
    sort(p,p+n);                      //对点排序：按 x 从小到大排序,如果 x 相同,按 y 排序
    n=unique(p,p+n)-p;                //去除重复点
    int v=0;
    //求下凸包.如果 p[i]是右拐弯的,这个点不在凸包上,往回退
    for(int i=0;i<n;i++){
        while(v>1 && sgn(Cross(ch[v-1]-ch[v-2],p[i]-ch[v-2]))<=0)
            v--;
        ch[v++]=p[i];
    }
    int j=v;
```

① 证明见《计算几何算法与应用(第 3 版)》，Mark de Berg 等著，邓俊辉译，清华大学出版社，8 页。

```
    //求上凸包
    for(int i = n - 2;i >= 0;i -- ){
        while(v > j && sgn(Cross(ch[v - 1] - ch[v - 2],p[i] - ch[v - 2]))<= 0)
            v -- ;
        ch[v++] = p[i];
    }
    if(n > 1) v -- ;
    return v;                         //返回值 v 是凸包的顶点数
}
int main(){
    int n;
    Point p[maxn],ch[maxn];           //输入点是 p[],凸包顶点放在 ch[]中
    while(scanf("%d",&n) && n){
        for(int i = 0;i < n;i++) scanf("%lf%lf",&p[i].x,&p[i].y);
        int v = Convex_hull(p,n,ch);  //返回凸包的顶点数 v
        double ans = 0;
        if(v == 1) ans = 0;
        else if(v == 2) ans = Distance(ch[0],ch[1]);
        else
            for(int i = 0;i < v;i++)  //计算凸包的周长
                ans += Distance(ch[i],ch[(i + 1) % v]);
        printf("%.2f\n",ans);
    }
    return 0;
}
```

【习题】

hdu 6325 “Interstellar Travel”。

11.1.6 最近点对

平面最近点对问题：给定平面上的 n 个点，找出距离最近的两个点。

先考虑暴力法，即列出所有的点对，然后比较每一对的距离，找出其中最短的。n 个点有 $c(n,2)$种组合，复杂度是 $O(n^2)$。

最近点对的标准算法是分治法，复杂度是 $O(n\log_2 n)$。下面是思路：

划分。把点的集合 S 平均分成两个子集 S_1 和 S_2(按点的 x 坐标排序，并按 x 的大小分成两半)，然后每个子集再划分成更小的两个子集，递归这个过程，直到子集中只有一个点或两个点。

解决。在每个子集中递归地求最近点对。

合并。在求出子集 S_1 和 S_2 的最接近点对后，合并 S_1 和 S_2。合并时有以下两种情况：

(1) 集合 S 中的最近点对在子集 S_1 内部或者 S_2 内部，那么可以简单地直接合并 S_1 和 S_2。

(2) 这两个点一个在 S_1 中，一个在 S_2 中，不能简单合并。设 S_1 中的最短距离是 d_1，S_2 中的最短距离是 d_2，在 S_1 和 S_2 的中间点 $p[\text{mid}]$附近找到所有离它小于 d_1 和 d_2 的点(仍

然按 x 坐标值计算距离)，记录在点集 tmp_p[]中，这样那么最近点对就在这些点中。这样在这些点中找最近点对就行了。用分治法求最近点对应如图 11.12 所示。但是，仍然不能直接用暴力法列出点集 tmp_p[]中的所有点对，否则会 TLE。可以先按 y 坐标值对 tmp_p[]的点排序(这次不能按 x 坐标值排序，请思考为什么)，然后用剪枝把不符合条件的去掉。具体见下面例题的代码。

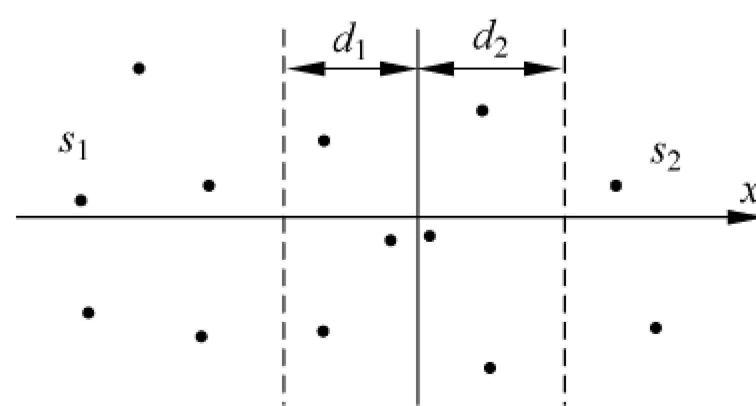


图 11.12　用分治法求最近点对

hdu 1007 "Quoit Design"

给定平面上的 n 个点，$2 \leqslant n \leqslant 100\,000$。

找到最近点对，输出最近点对距离的一半。

下面是代码，注意其中分治法和剪枝的内容。程序比较简单，读者应该能自己写出来。

```
#include <bits/stdc++.h>
using namespace std;
const double eps = 1e-8;
const int MAXN = 100010;
const double INF = 1e20;
int sgn(double x){
    if(fabs(x) < eps)   return 0;
    else return x<0?-1:1;
}
struct Point{
    double x,y;
};
double Distance(Point A, Point B){return hypot(A.x-B.x,A.y-B.y);}
bool cmpxy(Point A,Point B){                          //排序：先对横坐标 x 排序，再对 y 排序
  return sgn(A.x-B.x)<0 || (sgn(A.x-B.x) == 0 && sgn(A.y-B.y)<0);
}
bool cmpy(Point A,Point B){return sgn(A.y-B.y)<0;}   //只对 y 坐标排序
Point p[MAXN],tmp_p[MAXN];
double Closest_Pair(int left,int right){
    double dis = INF;
    if(left == right) return dis;                     //只剩一个点
    if(left + 1 == right)                             //只剩两个点
        return Distance(p[left], p[right]);
    int mid = (left+right)/2;                         //分治
    double d1 = Closest_Pair(left,mid);               //求 s1 内的最近点对
    double d2 = Closest_Pair(mid+1,right);            //求 s2 内的最近点对
    dis = min(d1,d2);
    int k = 0;
    for(int i=left;i<=right;i++)                      //在 s1 和 s2 中间附近找可能的最小点对
        if(fabs(p[mid].x - p[i].x) <=dis)             //按 x 坐标来找
            tmp_p[k++] = p[i];
  sort(tmp_p,tmp_p+k,cmpy);                   //按 y 坐标排序，用于剪枝.这里不能按 x 坐标排序
```

```
        for(int i = 0;i < k;i++)
            for(int j = i + 1;j < k;j++){
                if(tmp_p[j].y - tmp_p[i].y >= dis)  break;    //剪枝
                dis = min(dis,Distance(tmp_p[i],tmp_p[j]));
            }
    return dis;                                               //返回最小距离
}
int main(){
    int n;
    while(~scanf("%d",&n) && n){
        for(int i = 0;i < n;i++) scanf("%lf%lf",&p[i].x,&p[i].y);
        sort(p,p + n,cmpxy);                                  //先排序
        printf("%.2f\n",Closest_Pair(0,n - 1)/2);             //输出最短距离的一半
    }
    return 0;
}
```

【习题】

hdu 5721 “Palace”。

11.1.7 旋转卡壳

对于平面上的点集，可以用两条或更多平行线来“卡”住它们，从而解决很多问题。图 11.13 给出了一些应用场合。

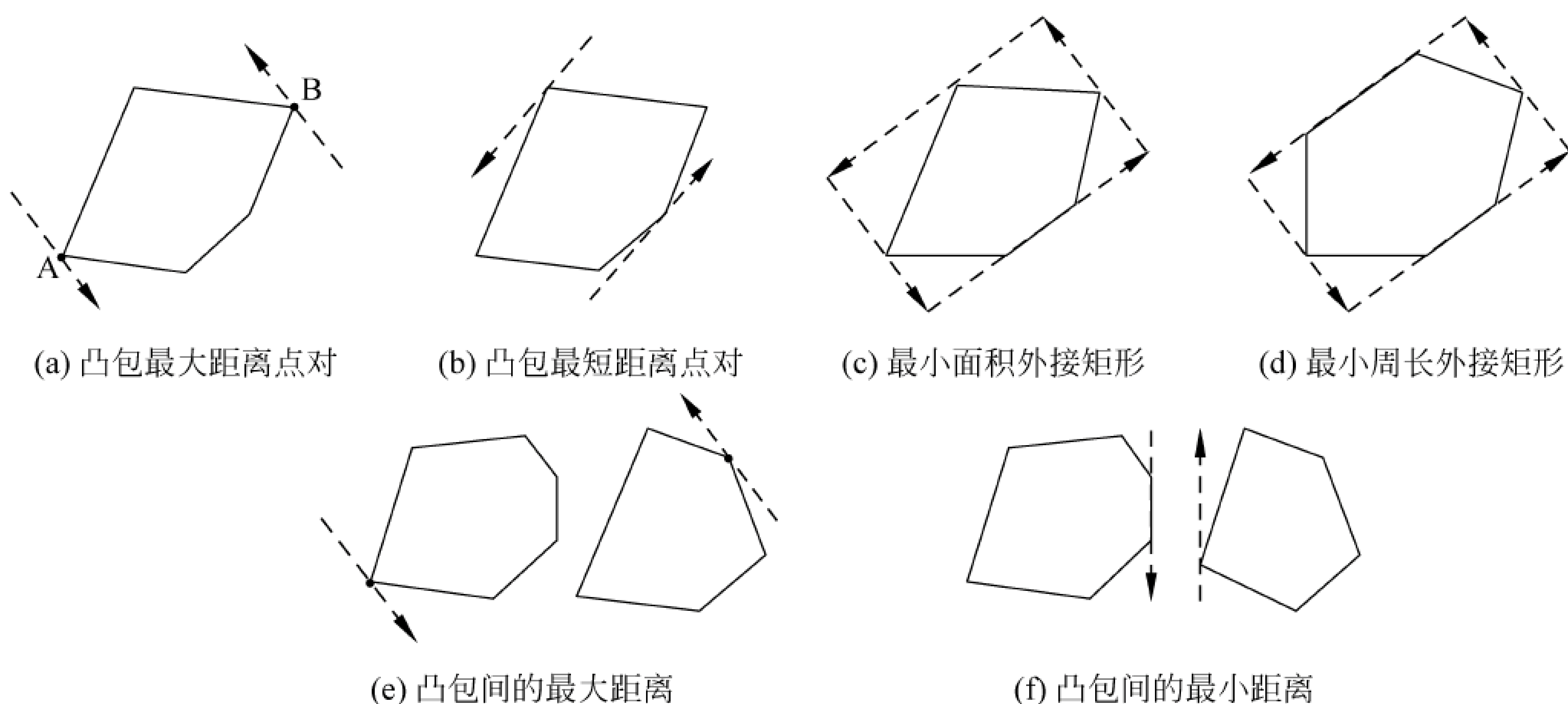


图 11.13 旋转卡壳的应用

两条平行线与凸包的交点称为对踵点对（antipodal pair），例如图 11.13(a)中的 A、B 点。找对踵点对，可以使用被形象地称为旋转卡壳（rotating calipers）的方法。

旋转卡壳算法是这样操作的：

(1) 找初始的对踵点对和平行线。可以取 y 坐标最大和最小的两个点，经过这两个点做两条水平线，一条向左，一条向右。

(2) 同时逆时针旋转两条线,直到其中一条线与多边形的一条边重合。此时得到新的对踵点对。如果题目要求最大距离点对,可以计算新对踵点对的距离,并比较和更新。

(3) 重复(2),直到回到初始对踵点。

【习题】

hdu 2202,凸包+旋转卡壳。

hdu 2187/2823。

hdu 5251,凸包+旋转卡壳求最小矩形覆盖。

11.1.8 半平面交

半平面就是平面的一半。

一个半平面用一条有向直线来定义。一条直线把平面分成两部分,为区分这两部分,这条直线应该是有向的,可以定义它左侧的平面是它代表的半平面。

给定一些半平面,它们相交会围成一片区域,例如图 11.14 所示的情况。

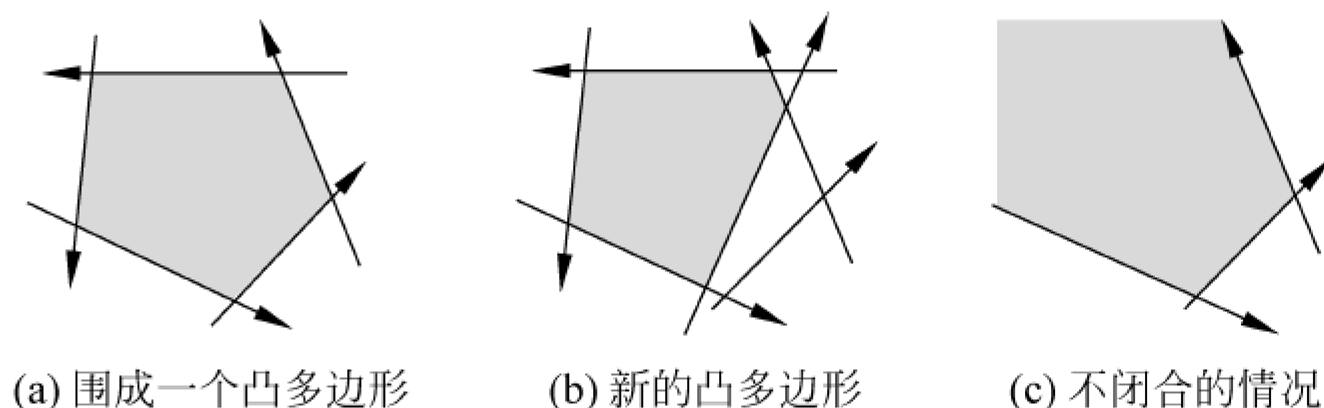


图 11.14 半平面交

图 11.14(a)中的 5 个半平面围成了一个凸多边形。如果再添加一个穿过凸多边形的半平面,那么凸多边形会变成图 11.14(b)。半平面交也可能不会闭合成一个凸多边形,而是成为图 11.14(c)的无边界的情况。在编程时为方便处理,可以在合适的地方人为添加半平面,闭合为凸多边形。

半平面的交一定是凸多边形(可能不闭合),所以求解半平面交问题就是求解形成的凸多边形。

1. 半平面的定义

表示半平面的有向直线,定义如下:

```
struct Line{
    Point p;                                    //直线上的一个点
    Vector v;                                   //方向向量,它的左边是半平面
    double ang;                                 //极角,从 x 正半轴旋转到 v 的角度
    Line(){};
    Line(Point p, Vector v):p(p),v(v){ang = atan2(v.y, v.x);}
    bool operator < (Line &L){return ang < L.ang;}   //用于排序
};
```

2. 半平面交算法

半平面交有一个显而易见的算法,即增量法,描述如下:

(1) 初始凸包。先人为设定一个极大的矩形,作为初始凸多边形,它能把最后形成的凸多边形包含进来。

(2) 逐一添加半平面,更新凸多边形。例如添加半平面 K,如果它能切割当前的凸多边形,则保留 K 左边的点,删除它右边的点,并把 K 与原凸多边形的交点加入到新的凸多边形中。

增量法不太好,它的复杂度是 $O(n^2)$,即一共有 n 次切割,每次切割都是 $O(n)$的。在下一页的例题 hdu 2297 中,$0<n\leqslant 50\ 000$,用增量法会 TLE。

下面介绍的算法,其复杂度为 $O(n\log_2 n)$。

思考半平面交最终形成的凸多边形,沿逆时针顺序看,它的边的极角(或者斜率)是单调递增的。那么,可以先按极角递增的顺序对半平面进行排序,然后逐个进行半平面交,最后就得到了凸多边形。在这个过程中,用一个双端队列记录构成凸多边形的半平面:队列的首部指向最早加入凸多边形的半平面,尾部指向新加入的半平面。

算法的具体步骤如下:

(1) 对所有半平面按极角排序。

(2) 初始时,加入第 1 个半平面,双端队列的首部和尾部都指向它。

(3) 逐个加入和处理半平面。图 11.15 演示了基本情况,原来半平面只有 1 和 2,加入半平面 3。注意,由于半平面已经排序,3 的极角比 1、2 大,所以有图 11.15 所示的 4 种情况。

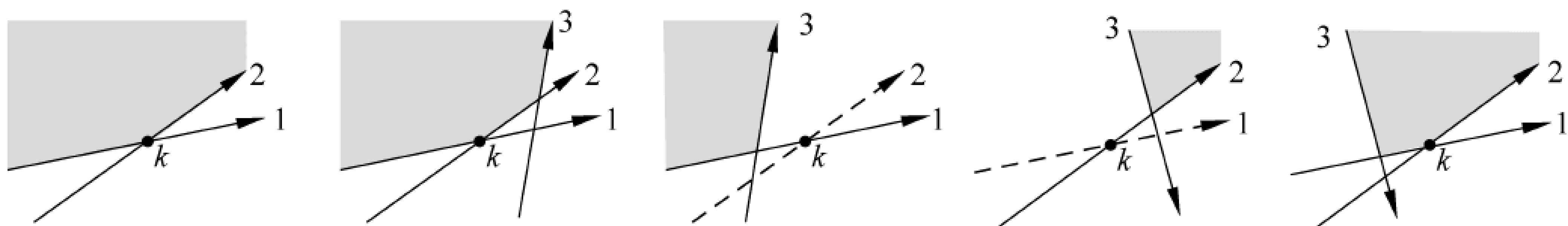
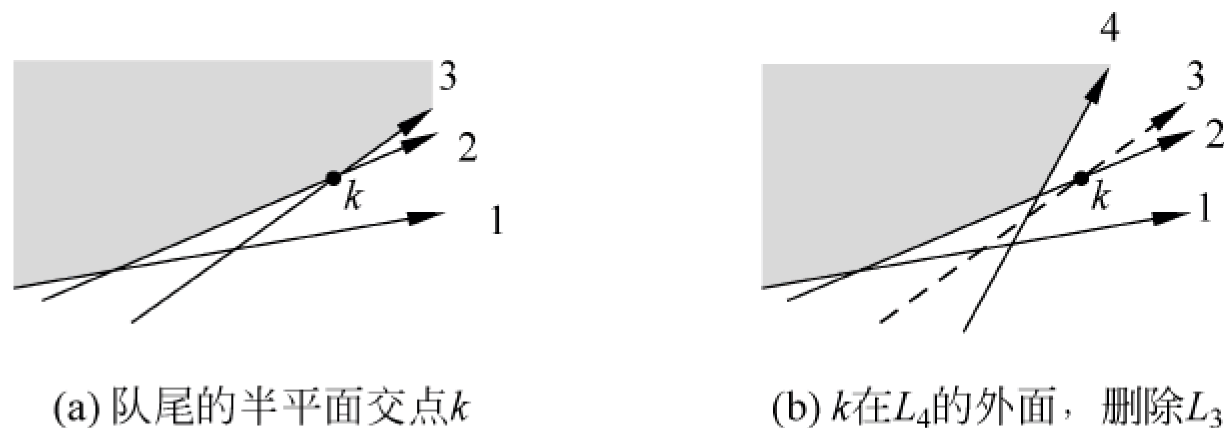


图 11.15　在半平面 1 和 2 上加入半平面 3 的 4 种情况

如果当前双端队列中不止有两个半平面,可以根据上面的讨论进行扩展。例如当前处理到半平面 L_i,有 4 种情况:L_i 可以直接加入队列;L_i 覆盖了原队尾;L_i 覆盖了原队首;L_i 不能加入到队列。下面讨论后面 3 种情况。

情况 1:L_i 覆盖原队尾。操作是:while 队尾的两个半平面的交点在 L_i 外面,那么删除队尾半平面。例如在图 11.16(a)中,队尾的两个半平面 L_2、L_3 的交点是 k。图(b)中新加入半平面 L_4,因为 k 在 L_4 的外面(点 k 在有向直线 L_4 的右边),删除队尾的半平面 L_3。



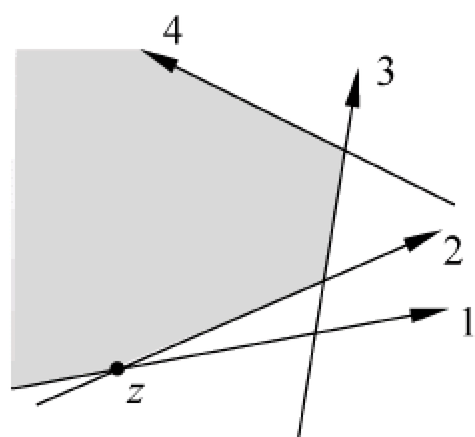
(a) 队尾的半平面交点k　　(b) k在L_4的外面,删除L_3

图 11.16　处理队尾

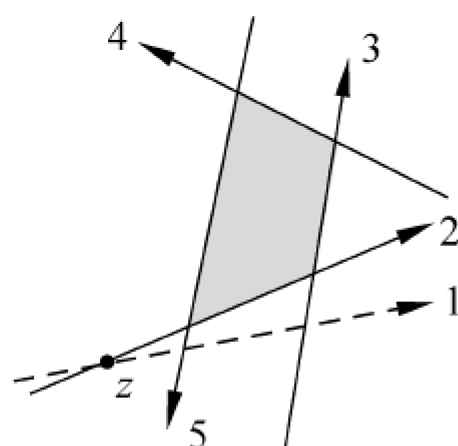
情况 2:L_i 覆盖原队首。操作是:while 队首的两个半平面的交点在 L_i 外面,那么删除队首的半平面。例如在图 11.17(a)中,队首 L_1、L_2 的两个半平面的交点是 z,图(b)中新加



入半平面 L_5，因为 z 在 L_5 的外面，删除队首的半平面 L_1。



(a) 队首半的平面交点z



(b) z在L_5的外面，删除L_1

图 11.17 处理队首

情况 3：L_i 不能加入到队列。例如图 11.18 所示的半平面 L_5，在步骤(3)中是合法的，但是它其实是无用的，不能加入到队列。判断条件是：尾部 L_4、L_5 的交点 r 在首部 L_1 的外面，则删除 L_5。

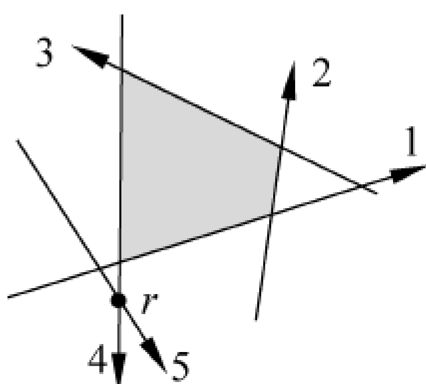


图 11.18 删除无用半平面 L_5

上述步骤的代码实现见下面的例题。

复杂度分析。排序，复杂度是 $O(n\log_2 n)$；逐个加入半平面，共检查 $O(n)$次，所以总复杂度是 $O(n\log_2 n)$。

3. 半平面的应用

下面的题目是很好的例子，它是半平面交的一个应用场景。

hdu 2297 "Run"

n 个人($0<n\leqslant 50\ 000$)在一条笔直的路上跑马拉松。设初始时每个人处于不同的位置，然后每个人都以自己的恒定速度不停地往前跑。

给定这 n 个人的初始位置和速度，问有多少人可能在某时刻成为第一？

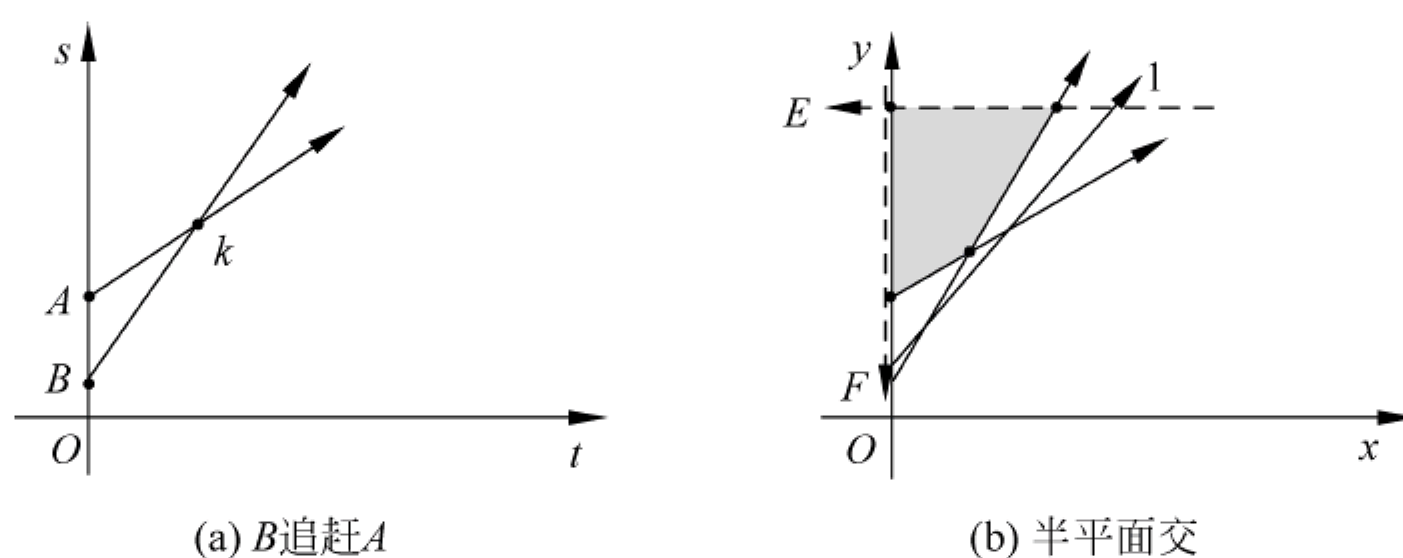
读者可以先思考：这一题如何建模为平面几何的半平面问题？

这一题实际上是半平面交的裸题，下面是建模过程。

以时间 t 为横轴，距离 s 为纵轴。设某人的初始位置在 A 点，从 A 出发画一条直线。他在某个时间段 Δt 内经过距离 Δs，两者的比值是直线的斜率，其物理意义正好是速度。他在某时刻 t'的位置就是他在这条直线上的纵坐标 s'。这条直线代表了他的运动轨迹。运动轨迹始终位于第一象限。

图 11.19(a)中的两条直线是两个人 A 和 B 的运动轨迹，交叉点 k 是 B 追上 A 的点。

如果有 n 个人，那么就有 n 条直线在第一象限，见图(b)。相交的点是追上的点，但追上后不一定排第一，例如图中的线 1，它与其他线有两个交点，但都不是第一。只有凸面上的点才是题目要求的排名第一的点。另外，由于这些直线的半平面交不是一个完整的凸多边形，为方便编程，可以加两个半平面 E 和 F，形成闭合的凸多边形，其中 E 是 y 值无穷大的向左的水平线，F 是反向的 y 轴。图中阴影是半平面交形成的凸多边形，凸多边形的顶点数量去掉最上面的两个黑点，就是题目要求的排过第一名的数量。



(a) B追赶A　　(b) 半平面交

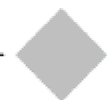
图 11.19　追赶问题

下面是 hdu 2297 的代码①。

```
#include<bits/stdc++.h>
using namespace std;
const double INF = 1e12;
const double pi = acos(-1.0);
const double eps = 1e-8;
int sgn(double x){
    if(fabs(x) < eps)  return 0;
    else return x<0?-1:1;
}
struct Point{
    double x,y;
    Point(){}
    Point(double x,double y):x(x),y(y){}
    Point operator + (Point B){return Point(x+B.x,y+B.y);}
    Point operator - (Point B){return Point(x-B.x,y-B.y);}
    Point operator * (double k){return Point(x*k,y*k);}
};
typedef Point Vector;
double Cross(Vector A,Vector B){return A.x*B.y - A.y*B.x;}  //叉积
struct Line{
    Point p;
    Vector v;
    double ang;
    Line(){};
    Line(Point p,Vector v):p(p),v(v){ang=atan2(v.y,v.x);}
    bool operator < (Line &L){return ang<L.ang;}               //用于极角排序
};
//点 p 在线 L 的左边,即点 p 在线 L 的外面
bool OnLeft(Line L,Point p){return sgn(Cross(L.v,p-L.p))>0;}
Point Cross_point(Line a,Line b){                              //两直线的交点
    Vector u=a.p-b.p;
    double t=Cross(b.v,u)/Cross(a.v,b.v);
    return a.p+a.v*t;
}
```

① 其中 HPI()的代码改编自《算法竞赛入门经典训练指南》,刘汝佳、陈锋著,清华大学出版社,278 页。

```
vector<Point> HPI(vector<Line> L){            //求半平面交,返回凸多边形
    int n = L.size();
    sort(L.begin(),L.end());                  //将所有半平面按照极角排序
    int first,last;                           //指向双端队列的第一个和最后一个元素
    vector<Point> p(n);                       //两个相邻半平面的交点
    vector<Line> q(n);                        //双端队列
    vector<Point> ans;                        //半平面交形成的凸包
    q[first = last = 0] = L[0];
    for(int i = 1;i<n;i++){
        //情况 1: 删除尾部的半平面
        while(first<last && !OnLeft(L[i], p[last-1])) last--;
        //情况 2: 删除首部的半平面
        while(first<last && !OnLeft(L[i], p[first]))  first++;
        q[++last] = L[i];                     //将当前的半平面加入双端队列的尾部
        //极角相同的两个半平面保留左边
        if(fabs(Cross(q[last].v,q[last-1].v)) < eps){
            last--;
            if(OnLeft(q[last],L[i].p)) q[last] = L[i];
        }
        //计算队列尾部的半平面交点
        if(first<last) p[last-1] = Cross_point(q[last-1],q[last]);
    }
    //情况 3: 删除队列尾部的无用半平面
    while(first<last && !OnLeft(q[first],p[last-1])) last--;
    if(last-first<=1) return ans;             //空集
    p[last] = Cross_point(q[last],q[first]);  //计算队列首尾部的交点
    for(int i = first;i<=last;i++)  ans.push_back(p[i]);  //复制
    return ans;                               //返回凸多边形
}
int main(){
    int T,n;
    cin>>T;
    while(T--){
        cin>>n;
        vector<Line> L;
        //加一个半平面 F:反向 y 轴
        L.push_back(Line(Point(0,0),Vector(0,-1)));
        //加一个半平面 E:y 极大的向左的直线
        L.push_back(Line(Point(0,INF),Vector(-1,0)));
        while(n--){
            double a,b;
            scanf("%lf%lf",&a,&b);
            L.push_back(Line(Point(0,a),Vector(1,b)));
        }
        vector<Point> ans = HPI(L);           //得到凸多边形
        printf("%d\n",ans.size()-2);          //去掉人为加的两个点
    }
    return 0;
}
```



【习题】

hdu 4316,凸包+半平面交。

hdu 3982,半平面交。

11.2　圆

11.2.1　基本计算

1. 圆的定义

用圆心和半径表示圆。

```
struct Circle{
    Point c;                                          //圆心
    double r;                                         //半径
    Circle(){}
    Circle(Point c,double r):c®,r®{}
    Circle(double x,double y,double _r){c = Point(x,y);r = _r;}
};
```

2. 点和圆的关系

点和圆的关系根据点到圆心的距离判断。

```
int Point_circle_relation(Point p, Circle C){
    double dst = Distance(p,C.c);
    if(sgn(dst - C.r) < 0) return 0;                  //0: 点在圆内
    if(sgn(dst - C.r) == 0) return 1;                 //1: 点在圆上
    return 2;                                         //2: 点在圆外
}
```

3. 直线和圆的关系

直线和圆的关系根据圆心到直线的距离判断。

```
int Line_circle_relation(Line v,Circle C){
    double dst = Dis_point_line(C.c,v);
    if(sgn(dst - C.r) < 0) return 0;                  //0: 直线和圆相交
    if(sgn(dst - C.r) == 0) return 1;                 //1: 直线和圆相切
    return 2;                                         //2: 直线在圆外
}
```

4. 线段和圆的关系

线段和圆的关系根据圆心到线段的距离判断。

```
int Seg_circle_relation(Segment v,Circle C){
    double dst = Dis_point_seg(C.c,v);
```

```
    if(sgn(dst - C.r) < 0) return 0;                   //0: 线段在圆内
    if(sgn(dst - C.r)  == 0) return 1;                 //1: 线段和圆相切
    return 2;                                          //2: 线段在圆外
}
```

5. 直线和圆的交点

求直线和圆的交点可以按图 11.20 所示,先求圆心 c 在直线上的投影 q,再求距离 d,然后根据 r 和 d 求出长度 k,最后求出两个交点 $p_a = q + n * k$、$p_b = q - n * k$,其中 k 是直线的单位向量。

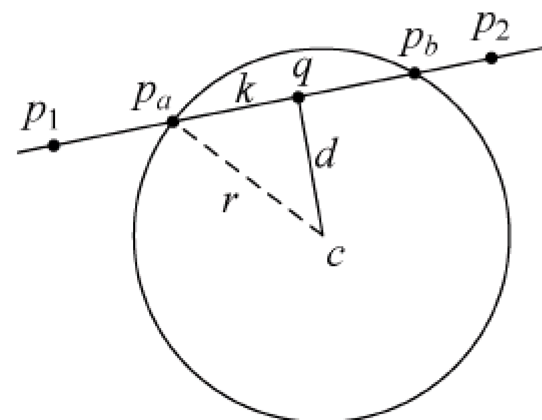


图 11.20　直线和圆的交点

```
//pa、pb 是交点.返回值是交点的个数
int Line_cross_circle(Line v,Circle C,Point &pa,Point &pb){
    if(Line_circle_relation(v, C) == 2)  return 0;  //无交点
    Point q  =  Point_line_proj(C.c,v);             //圆心在直线上的投影点
    double d  =  Dis_point_line(C.c,v);             //圆心到直线的距离
    double k  =  sqrt(C.r * C.r - d * d);
    if(sgn(k)  ==  0){                              //一个交点,直线和圆相切
        pa  =  q;pb  =  q;return 1;
    }
    Point n = (v.p2 - v.p1)/ Len(v.p2 - v.p1);      //单位向量
    pa  =  q  +  n * k;  pb  =  q  -  n * k;
    return 2;                                       //两个交点
}
```

6. 模板的使用

下面用 hdu 5572 题演示点、线、圆的几何模板的使用,如图 11.21 所示。这一题出自 2015 年 ACM-ICPC 区域赛上海赛区的现场赛,题目的详细说明见 12.2.4 节。

> **hdu 5572 "An Easy Physics Problem"**
>
> 在一个无限光滑的桌面上有一个固定的大圆柱体,还有一个体积忽略不计的小球。开始时球静止于 A 点,给它一个初始速度和方向,如果球撞到圆柱体,它会弹回,没有能量损失。经过一段时间,小球是否会经过 B 点?

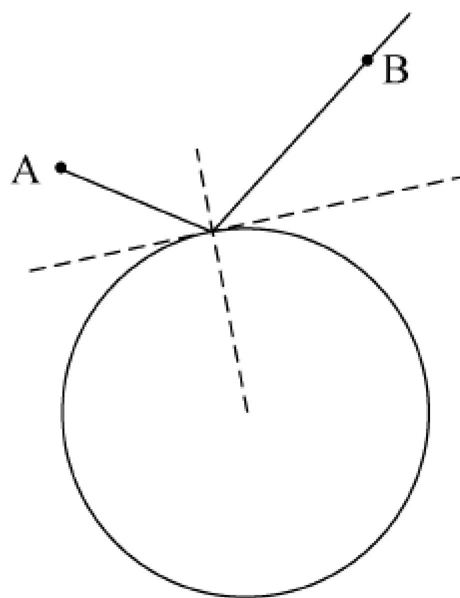


图 11.21　hdu 5572 题图示

这是一道中等题，考核参赛人员对几何基本模板的使用。该题目的逻辑很简单，但是综合性较强，它涉及的计算几何知识有直线的表示、圆的表示、点在直线上的投影、点到直线的距离、点对于直线的镜像点、线段和圆的关系、直线和圆的交点等。

下面的代码完全套用了前面给出的模板。

```
#include <bits/stdc++.h>
using namespace std;
const double eps = 1e-8;                          //本题如果设定 eps = 1e-10,会 Wrong Answer
int sgn(double x){                                //判断 x 是否等于 0
    if(fabs(x) < eps)  return 0;
    else return x<0?-1:1;
}
struct Point{                                     //定义点及其基本运算
    double x,y;
    Point(){}
    Point(double x,double y):x(x),y(y){}
    Point operator + (Point B){return Point(x+B.x,y+B.y);}
    Point operator - (Point B){return Point(x-B.x,y-B.y);}
    Point operator * (double k){return Point(x*k,y*k);}
    Point operator / (double k){return Point(x/k,y/k);}
};
typedef Point Vector;                             //定义向量
double Dot(Vector A,Vector B){return A.x*B.x + A.y*B.y;}    //点积
double Len(Vector A){return sqrt(Dot(A,A));}      //向量的长度
double Len2(Vector A){return Dot(A,A);}           //向量长度的平方
double Cross(Vector A,Vector B){return A.x*B.y - A.y*B.x;}  //叉积
double Distance(Point A, Point B){return hypot(A.x-B.x,A.y-B.y);}
struct Line{
    Point p1,p2;
    Line(){}
    Line(Point p1,Point p2):p1(p1),p2(p2){}
};
typedef Line Segment;                             //定义线段,两端点是 p1、p2
int Point_line_relation(Point p,Line v){
    int c = sgn(Cross(p-v.p1,v.p2-v.p1));
    if(c<0)return 1;                              //1: p 在 v 的左边
    if(c>0)return 2;                              //2: p 在 v 的右边
    return 0;                                     //0: p 在 v 上
}
double Dis_point_line(Point p, Line v){           //点到直线的距离
    return fabs(Cross(p-v.p1,v.p2-v.p1))/Distance(v.p1,v.p2);
}
//点到线段的距离
double Dis_point_seg(Point p, Segment v){
    if(sgn(Dot(p-v.p1,v.p2-v.p1))<0 || sgn(Dot(p-v.p2,v.p1-v.p2))<0)
        return min(Distance(p,v.p1),Distance(p,v.p2));
    return Dis_point_line(p,v);
}
//点在直线上的投影
```

```
Point Point_line_proj(Point p, Line v){
    double k = Dot(v.p2 - v.p1,p - v.p1)/Len2(v.p2 - v.p1);
    return v.p1 + (v.p2 - v.p1) * k;
}
//点 p 对直线 v 的对称点
Point Point_line_symmetry(Point p, Line v){
    Point q  =  Point_line_proj(p,v);
    return Point(2 * q.x - p.x,2 * q.y - p.y);
}
struct Circle{
    Point c;                                          //圆心
    double r;                                         //半径
    Circle(){}
    Circle(Point c,double r):c(c),r(r){}
    Circle(double x,double y,double _r){c = Point(x,y);r  =  _r;}
};
//线段和圆的关系：0 为线段在圆内,1 为线段和圆相切,2 为线段在圆外
int Seg_circle_relation(Segment v,Circle C){
    double dst  =  Dis_point_seg(C.c,v);
    if(sgn(dst - C.r) < 0) return 0;
    if(sgn(dst - C.r)  == 0) return 1;
    return 2;
}
//直线和圆的关系：0 为直线在圆内,1 为直线和圆相切,2 为直线在圆外
int Line_circle_relation(Line v,Circle C){
    double dst  =  Dis_point_line(C.c,v);
    if(sgn(dst - C.r) < 0) return 0;
    if(sgn(dst - C.r)  == 0) return 1;
    return 2;
}
//直线和圆的交点,pa、pb 是交点.返回值是交点的个数
int Line_cross_circle(Line v,Circle C,Point &pa,Point &pb){
    if(Line_circle_relation(v, C) == 2) return 0;     //无交点
    Point q  =  Point_line_proj(C.c,v);               //圆心在直线上的投影点
    double d  =  Dis_point_line(C.c,v);               //圆心到直线的距离
    double k  =  sqrt(C.r * C.r - d * d);
    if(sgn(k)  ==  0){                                //一个交点,直线和圆相切
        pa  =  q;   pb  =  q;   return 1;
    }
    Point n = (v.p2 - v.p1)/ Len(v.p2 - v.p1);        //单位向量
    pa  =  q  +  n * k;
    pb  =  q  -  n * k;
    return 2;                                         //两个交点
}
int main() {
    int T; scanf("%d", &T);
    for (int cas  =  1; cas <= T; cas++) {
        Circle O; Point A,B,V;
        scanf("%lf%lf%lf", &O.c.x, &O.c.y, &O.r);
        scanf("%lf%lf%lf%lf", &A.x, &A.y, &V.x, &V.y);
        scanf("%lf%lf", &B.x, &B.y);
```

```
        Line l(A, A + V);                              //射线
        Line t(A, B);
            //情况 1: 直线和圆不相交,而且直线经过点
        if(Point_line_relation(B,l) == 0
            && Seg_circle_relation(t,O)>= 1 && sgn(Cross(B - A,V)) == 0)
            printf("Case # %d: Yes\n", cas);
        else{
            Point pa, pb;                              //直线和圆的交点
            //情况 2: 直线和圆相切,不经过点
            if(Line_cross_circle(l,O,pa,pb) != 2)
                printf("Case # %d: No\n",cas);
            //情况 3: 直线和圆相交
            else{
                Point cut;                             //直线和圆的碰撞点
                if(Distance(pa,A) > Distance(pb,A)) cut = pb;
                else cut = pa;
                Line mid(cut, O.c);                    //圆心到碰撞点的直线
                Point en = Point_line_symmetry(A,mid); //镜像点
                Line light(cut, en);                   //反射线
                if(Distance(light.p2,B) > Distance(light.p1,B))
                    swap(light.p1, light.p2);
                if(sgn(Cross(light.p2 - light.p1,
                          Point(B.x - cut.x,B.y - cut.y))) == 0)
                printf("Case # %d: Yes\n", cas);
                else
                    printf("Case # %d: No\n", cas);
            }
        }
    }
    return 0;
}
```

11.2.2 最小圆覆盖

最小圆覆盖问题：给定 n 个点的平面坐标，求一个半径最小的圆，把 n 个点全部包围，部分点在圆上。

常见的算法有两种，即几何算法和模拟退火算法。

1. 几何算法

这个最小圆可以由 n 个点中的两个点或 3 个点确定。由两点定圆时，圆心是线段 AB 的中点，半径是 AB 长度的一半，其他点都在这个圆内；如果两点不足以包围所有点，就需要三点定圆，此时圆心是 A、B、C 这 3 个点组成的三角形的外心，如图 11.22 所示。

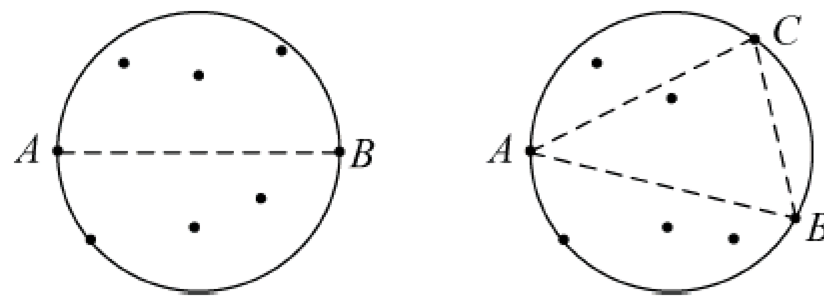


图 11.22 两点定圆或三点定圆

最小覆盖圆的获得就是寻找能两点定圆或三点定圆的那几个点。

一般用增量法求最小圆覆盖。算法从一个点开始，每次加入一个新的点，则更新最小圆，直到扩展到全部 n 个点。设前 i 个点的最小覆盖圆是 C_i，过程如下：

(1) 加第 1 个点 p_1。C_1 的圆心就是 p_1，半径为 0。

(2) 加第 2 个点 p_2。新的 C_2 的圆心是线段 p_1p_2 的中心，半径为两点距离的一半。这一步操作是两点定圆。

(3) 加第 3 个点 p_3。有两种情况：p_3 在 C_2 的内部或圆周上，不影响原来的最小圆，忽略 p_3；p_3 在 C_2 的外部，此时 C_2 已不能覆盖所有 3 个点，需要更新。下面讨论 p_3 在 C_2 外部的情况。因为 p_3 一定在新的 C_3 上，现在的任务转换为在 p_1、p_2 中找一个点或两个点，与 p_3 一起两点定圆或三点定圆。重新定圆的过程相当于回到第(1)步，把 p_3 作为第 1 个点加入，然后再加入 p_1、p_2。

(4) 加第 4 个点 p_4。分析和步骤(3)类似，为加强理解，这里重复说明一次。如果 p_4 在 C_3 的内部或圆周上，忽略它。如果在 C_3 的外部，那么需要求新的最小圆，此时 p_4 肯定在新的 C_4 的圆周上。任务转换为在 p_1、p_2、p_3 中找一个点或两个点，与 p_4 一起构成最小圆。先检查能不能找到一个点，用两点定圆；如果两点不够，就找到第 3 个点，用三点定圆。重新定圆的过程和前 3 个步骤类似，即把 p_4 作为第 1 个点加入，然后加入 p_1、p_2、p_3。

(5) 持续进行下去，直到加完所有点。

算法的思路概括如下：

假设已经求得前 $i-1$ 个点的 C_{i-1}，现在加入第 i 个点，有两种情况。

(1) i 在 C_{i-1} 的内部或圆周上，忽略 i。

(2) i 在 C_{i-1} 的外部，需要求新的 C_i。首先，i 肯定在 C_i 上，然后重新把前面的 $i-1$ 个点依次加入，根据两点定圆或者三点定圆重新构造最小圆。

几何算法的复杂度分析。在下面的例题中给出了模板代码。其中有 3 层 for 循环，看起来似乎是 $O(n^3)$。不过，如果点的分布是随机的，用概率进行分析可以得出程序的复杂度是接近 $O(n)$ 的。在下面的代码中，用 random_shuffle()函数进行随机打乱。

例如，如果前两个点 p_1 和 p_2 恰好就是最后的两点定圆，那么其他的所有点都只需要检查一次是否在 C_2 内就行了，程序在第一层 for 就结束了。对于算法复杂度的详细证明，请读者查阅有关资料。

下面的例题是最小圆覆盖的裸题。

hdu 3007 "Buried memory"

输入 n 个点的坐标，$n<500$，求最小圆覆盖。

代码如下：

```
#include<bits/stdc++.h>
using namespace std;
#define eps 1e-8
const int maxn = 505;
int sgn(double x){
    if(fabs(x) < eps)  return 0;
    else return x<0? -1:1;
```

```
}
struct Point{
    double x, y;
};
double Distance(Point A, Point B){return hypot(A.x - B.x,A.y - B.y);}
//求三角形 abc 的外接圆的圆心
Point circle_center(const Point a, const Point b, const Point c){
    Point center;
    double a1 = b.x - a.x, b1 = b.y - a.y, c1 = (a1 * a1 + b1 * b1)/2;
    double a2 = c.x - a.x, b2 = c.y - a.y, c2 = (a2 * a2 + b2 * b2)/2;
    double d = a1 * b2 - a2 * b1;
    center.x = a.x + (c1 * b2 - c2 * b1)/d;
    center.y = a.y + (a1 * c2 - a2 * c1)/d;
    return center;
}
//求最小覆盖圆,返回圆心 c、半径 r:
void min_cover_circle(Point *p, int n, Point &c, double &r){
    random_shuffle(p, p + n);                   //随机函数,打乱所有点.这一步很重要
    c = p[0]; r = 0;                            //从第 1 个点 p0 开始.圆心为 p0,半径为 0
    for(int i = 1;i < n;i++)                    //扩展所有点
        if(sgn(Distance(p[i],c) - r)> 0){       //点 pi 在圆的外部
            c = p[i]; r = 0;                    //重新设置圆心为 pi,半径为 0
            for(int j = 0;j < i;j++)            //重新检查前面所有的点
                if(sgn(Distance(p[j],c) - r)> 0){ //两点定圆
                    c.x = (p[i].x + p[j].x)/2;
                    c.y = (p[i].y + p[j].y)/2;
                    r = Distance(p[j],c);
                    for(int k = 0;k < j;k++)
                        if (sgn(Distance(p[k],c) - r)> 0){  //两点不能定圆就三点定圆
                            c = circle_center(p[i],p[j],p[k]);
                            r = Distance(p[i], c);
                        }
                }
        }
}
int main(){
    int n;                                      //点的个数
    Point p[maxn];                              //输入点
    Point c; double r;                          //最小覆盖圆的圆心和半径
    while(~scanf("%d",&n) && n){
        for(int i = 0;i < n;i++) scanf("%lf %lf",&p[i].x,&p[i].y);
        min_cover_circle(p,n,c,r);
        printf("%.2f %.2f %.2f\n",c.x,c.y,r);
    }
    return 0;
}
```

2. 模拟退火算法

如果题目的数据规模不大,最小圆覆盖还可以用模拟退火算法实现,请读者先回顾第 6

章的“6.1.4 模拟退火”。用模拟退火求最小圆,不断迭代(降温);在每次迭代时,找到能覆盖到所有点的一个圆;在多次迭代中,逐步逼近最后要求的圆心和半径。

下面的函数 min_cover_circle()是模拟退火程序,用它替换上面的同名函数即可。

```
void min_cover_circle(Point *p, int n, Point &c, double &r){
    double T = 100.0;                       //初始温度
    double delta = 0.98;                    //降温系数
    c = p[0];
    int pos;
    while (T > eps){                        //eps 是终止温度
        pos = 0; r = 0;                     //初始:p[0]是圆心,半径是 0
        for(int i = 0; i <= n - 1; i++)     //找距圆心最远的点
            if (Distance(c, p[i]) > r){
                r = Distance(c, p[i]);      //距圆心最远的点肯定在圆周上
                pos = i;
            }
        c.x += (p[pos].x - c.x) / r * T;    //逼近最后的解
        c.y += (p[pos].y - c.y) / r * T;
        T *= delta;
    }
}
```

模拟退火的程序很简单,不过需要仔细选择初始温度 T、降温系数 delta、终止温度 eps 等,程序的复杂度也和它们有关。在本题中,模拟退火算法的复杂度远高于几何算法。OJ 返回的 AC 时间,几何算法是 46ms,模拟退火是 670ms。


视频讲解

【习题】

hdu 2215,最小圆覆盖。

11.3 三维几何

11.3.1 三维点和向量

1. 点和向量

在三维几何中,点和向量的表示和二维几何是类似的。同样也可以定义三维空间的运算。

```
struct Point3{                              //三维点
    double x,y,z;
    Point3(){}
    Point3(double x,double y,double z):x(x),y(y),z(z){}
    Point3 operator + (Point3 B){return Point3(x+B.x,y+B.y,z+B.z);}
    Point3 operator - (Point3 B){return Point3(x-B.x,y-B.y,z-B.z);}
    Point3 operator * (double k){return Point3(x*k,y*k,z*k);}
```

```
    Point3 operator / (double k){return Point3(x/k,y/k,z/k);}
    bool operator == (Point3 B){
        return sgn(x - B.x) == 0 && sgn(y - B.y) == 0 && sgn(z - B.z) == 0;}
};
typedef Point3 Vector3;                          //三维向量
```

点和点的距离[①]:

```
double Distance(Vector3 A,Vector3 B){
    return sqrt((A.x - B.x) * (A.x - B.x) +
              (A.y - B.y) * (A.y - B.y) +
              (A.z - B.z) * (A.z - B.z));      }
```

2. 线和线段

和二维一样,三维的直线和线段也用两点定义。

```
struct Line3{
    Point3 p1,p2;
    Line3(){}
    Line3(Point3 p1,Point3 p2):p1(p1),p2(p2){}
};
typedef Line3 Segment3;                          //定义线段,两端点是 Point p1,p2
```

11.3.2 三维点积

1. 点积

三维点积的定义和二维的类似,定义如下:

$$\boldsymbol{A} \cdot \boldsymbol{B} = |\boldsymbol{A}||\boldsymbol{B}|\cos\theta$$

求向量 $\boldsymbol{A}$、$\boldsymbol{B}$ 点积的代码如下:

```
double Dot(Vector3 A,Vector3 B){return A.x * B.x + A.y * B.y + A.z * B.z;}
```

2. 点积的基本应用

和二维点积一样,三维点积有以下基本应用:

1) 判断向量 $\boldsymbol{A}$ 与 $\boldsymbol{B}$ 的夹角是钝角还是锐角

点积有正负,利用正负号可以判断向量的夹角:

若 $\mathrm{dot}(\boldsymbol{A},\boldsymbol{B})>0$,$\boldsymbol{A}$ 与 $\boldsymbol{B}$ 的夹角为锐角;

若 $\mathrm{dot}(\boldsymbol{A},\boldsymbol{B})<0$,$\boldsymbol{A}$ 与 $\boldsymbol{B}$ 的夹角为钝角;

若 $\mathrm{dot}(\boldsymbol{A},\boldsymbol{B})=0$,$\boldsymbol{A}$ 与 $\boldsymbol{B}$ 的夹角为直角。

2) 求向量 $\boldsymbol{A}$ 的长度

```
double Len(Vector3 A){return sqrt(Dot(A, A));}
```

或者是求长度的平方,避免开方运算:

```
double Len2(Vector3 A){return Dot(A, A);}
```

---

① 读者可能注意到,本章给出的三维函数和二维函数很多是重名的,例如这里的 Distance()。C++ 允许函数重载,所以即使在同一个程序中用同名来定义不同的函数也是允许的。而且建议重载函数,这样做可以简化编程。

3）求向量 **A** 与 **B** 的夹角大小

```
double Angle(Vector3 A,Vector3 B){return acos(Dot(A,B)/Len(A)/Len(B));}
```

11.3.3 三维叉积

二维叉积是一个带正负的数值，而三维叉积是一个向量。可以把三维向量 **A**、**B** 的叉积看成垂直于 **A** 和 **B** 的向量，如图 11.23 所示，其方向符合“右手定则”。

三维叉积的计算和二维叉积相似，不同的是计算后返回一个向量：

```
Vector3 Cross(Vector3 A,Vector3 B){
    return Point3(A.y * B.z - A.z * B.y, A.z * B.x - A.x * B.z, A.x * B.y -
A.y * B.x);
}
```

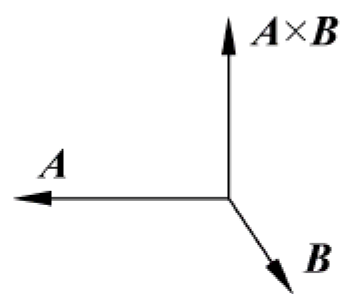


图 11.23　三维叉积

1. 三角形面积

三维的三角形面积计算和二维的相似，也是有向面积。先求三维叉积，然后取叉积的长度值。

```
//三角形面积的两倍
double Area2(Point3 A,Point3 B,Point3 C){return Len(Cross(B - A, C - A));}
```

判断点 p 是否在三角形 ABC 内，可以用 Area2()来计算。如果点 p 在三角形内部，那么用点 p 对三角形 ABC 进行三角剖分，形成的 3 个三角形的面积和与直接算 ABC 的面积，两者应该相等：

```
Dcmp(Area2(p,A,B) + Area2(p,B,C) + Area2(p,C,A), Area2(A,B,C)) == 0
```

2. 点和线的有关问题

点到直线的距离、点是否在直线上、点到线段的距离、点在直线上的投影等问题的代码和二维几何相似，见本章 11.4 节的几何模板。

3. 平面

用 3 个点可以确定一个平面。

```
struct Plane{
    Point3 p1,p2,p3;                          //平面上的 3 个点
    Plane(){}
    Plane(Point3 p1,Point3 p2,Point3 p3):p1(p1),p2(p2),p3(p3){}
};
```

4. 平面法向量

平面法向量是垂直于平面的向量，在平面问题中非常重要。它用叉积的概念计算即可，代码如下：

```
Point3 Pvec(Point3 A, Point3 B, Point3 C){return Cross(B - A,C - A);}
```

或者：

```
Point3 Pvec(Plane f){return Cross(f.p2 - f.p1,f.p3 - f.p1);}
```

5. 平面的有关问题

四点共平面、两平面平行、两平面垂直等问题的代码见本章11.4节的几何模板。

6. 直线和平面的交点

直线和平面有3种关系，即直线在平面上、直线和平面平行、直线和平面有交点。

一个平面，可以用平面 f 上的一点 $f.p1$ 以及平面的法向量 v 来决定。直线 u 用两点 $u.p1$ 和 $u.p2$ 决定。

下面的函数计算直线与平面的交点，交点是 p，函数的返回值是交点的个数。

```
int Line_cross_plane(Line3 u,Plane f,Point3 &p){
    Point3 v = Pvec(f);                        //平面的法向量
    double x = Dot(v, u.p2 - f.p1);
    double y = Dot(v, u.p1 - f.p1);
    double d = x - y;
    if(sgn(x) == 0 && sgn(y) == 0) return -1;//-1: v在f上
    if(sgn(d) == 0) return 0;                  //0: v与f平行
    p = ((u.p1 * x) - (u.p2 * y))/d;           //1: v与f相交
    return 1;
}
```

下面解释代码的正确性。

代码中的 v 是平面的法向量，它不一定是单位法向量，不过这里把它看成是单位法向量，不影响后续推理的正确性。$x=\text{Dot}(v,u.p2-f.p1)$ 是 $u.p2$ 到平面 f 的距离，$y=\text{Dot}(v,u.p1-f.p1)$ 是 $u.p1$ 到平面的距离。如果 $x=y=0$，说明直线在平面上；如果 $x=y\neq0$，即直线上的两点到平面的距离相等，说明直线和平面平行。

如果直线和平面相交，如何计算交点？

如图11.24所示，在 x、y、z 轴的任何一个方向上都有 $\frac{p-p_1}{p-p_2}=\frac{y}{x}$，推导得 $p=\frac{p_1 * x-p_2 * y}{x-y}$。

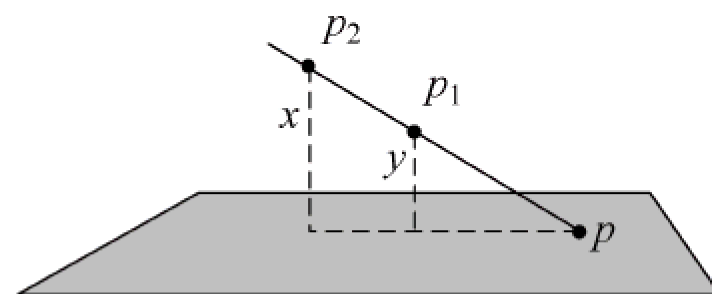


图11.24　直线和平面的交点

7. 四面体的有向体积

四面体是最简单的立体结构。四面体的体积等于底面三角形面积乘以高的1/3，利用叉积和点积很容易计算，代码如下：

```
//四面体有向体积×6
double volume4(Point3 a,Point3 b,Point3 c,Point3 d){
    return Dot(Cross(b - a,c - a),d - a); }
```

【习题】

hdu 1140/4617。

hdu 5733，四面体。

11.3.4 最小球覆盖

最小球覆盖问题：给定 n 个点的三维坐标，求一个半径最小的球，把 n 个点全部包围进来。

和最小圆覆盖一样，最小球覆盖问题也有两种解法，即几何算法和模拟退火算法。

1. 模拟退火算法

如果数据规模较小，可以用模拟退火算法求最小球覆盖。其代码和最小圆覆盖的程序几乎一样，只需加上对坐标 z 的处理即可。

2. 几何算法

和最小圆覆盖增量法的思路类似，最小球覆盖也可以由一些点来确定。一个三维空间中的球，需要 1～4 个点来确定。可以从一个点开始，每次加入一个新的点，更新最小球，直到扩展到全部 n 个点。设前 i 个点的最小覆盖球是 C_i，简单说明如下：

(1) 1 个点。C_1 的球心就是 p_1，半径为 0。

(2) 2 个点。新的 C_2 的球心是线段 p_1p_2 的中心，半径为两点距离的一半。

(3) 3 个点。3 个点构成的平面一定是球的大圆所在的平面，所以球心是三角形的外心，半径就是球心到某个点的距离。

(4) 4 个点。若 4 个点共面则转化到(3)，考虑某 3 个点的情况，若 4 点不共面，四面体可以唯一确定一个外接球。

(5) 对于 5 个及以上点，其最小球必为其中某 4 个点的外接球。

最小覆盖球的代码比较复杂。读者可以通过下面的例题来了解最小覆盖球问题的几何算法。

poj 2069 "Super Star"

输入 n 个点的三维坐标，$4\leqslant n\leqslant 30$，求最小球覆盖，输出球的半径。

11.3.5 三维凸包

三维凸包问题：给定三维空间的一些点，找到包含这些点的最小凸多面体。三维凸包问题是二维凸包问题的扩展，它是一个比较难的问题。

如果用暴力法求三维凸包，可以枚举任意 3 个点组成的三角形，判断其他点是否都在三角形构成的平面的一侧，如果是，则这个三角形是凸包的一个面。

三维凸包的常用算法是增量法。该算法的思想和最小圆覆盖的增量法有些类似，即把点一个个加入到凸包中。首先找到 4 个不共线、不共面的点，一起构成一个四面体，这是初始凸包，然后依次检查其他点，看这个点是否能在原凸包的基础上构成新的凸包。例如，当检查到点 p_i 时有两种情况：

(1) 如果 p_i 在当前的凸包内，忽略它。

(2) 如果 p_i 不在凸包内，说明用 p_i 可以更新凸包。具体做法是从 p_i 点向凸包看去，将能看到的面全部删除，并把 p_i 和留下的轮廓组合成新的面，填补被删除的面。



三维凸包题目的相关问题有凸包有几个表面、凸包的表面积、凸包的重心等。

复杂度。如果给定的点是随机排列的，算法的期望时间是 $O(n\log_2 n)$[①]的。

下面用一个例题给出三维凸包的模板代码。

hdu 3662 "3D Convex Hull"

输入 n 个点的三维坐标，求三维凸包有几个面。

代码如下[②]：

```
#include<bits/stdc++.h>
using namespace std;
const int MAXN = 1050;
const double eps = 1e-8;
struct Point3{                                           //三维：点
    double x,y,z;
    Point3(){}
    Point3(double x,double y,double z):x(x),y(y),z(z){}
    Point3 operator + (Point3 B){return Point3(x+B.x,y+B.y,z+B.z);}
    Point3 operator - (Point3 B){return Point3(x-B.x,y-B.y,z-B.z);}
    Point3 operator * (double k){return Point3(x*k,y*k,z*k);}
    Point3 operator / (double k){return Point3(x/k,y/k,z/k);}
};
typedef Point3 Vector3;
double Dot(Vector3 A,Vector3 B){return A.x*B.x+A.y*B.y+A.z*B.z;}
Point3 Cross(Vector3 A,Vector3 B){
    return Point3(A.y*B.z-A.z*B.y,A.z*B.x-A.x*B.z,A.x*B.y-A.y*B.x);}
double Len(Vector3 A){return sqrt(Dot(A,A));}            //向量的长度
double Area2(Point3 A,Point3 B,Point3 C){return Len(Cross(B-A, C-A));}
//四面体有向体积×6
double volume4(Point3 A,Point3 B,Point3 C,Point3 D){
    return Dot(Cross(B-A,C-A),D-A);}
struct CH3D{
    struct face{
        int a,b,c;                                       //凸包一个面上的3个点的编号
        bool ok;                                         //该面是否在最终凸包上
    };
    int n;                                               //初始顶点数
    Point3 P[MAXN];                                      //初始顶点
    int num;                                             //凸包表面的三角形数
    face F[8*MAXN];                                      //凸包表面的三角形
    int g[MAXN][MAXN];                                   //点i到点j属于哪个面
    //点在面的同向
    double dblcmp(Point3 &p,face &f){
        Point3 m = P[f.b] - P[f.a];
        Point3 n = P[f.c] - P[f.a];
```

① 证明见《计算几何算法与应用(第3版)》，Mark de Berg 等著，邓俊辉译，清华大学出版社，257页。

② 此代码中的 CH3D()是流传很广的经典模板，如果 CH3D()的原作者看到这里，请联系本书作者。

```
        Point3 t = p - P[f.a];
        return Dot(Cross(m,n),t);
    }
    void deal(int p,int a,int b){
        int f = g[a][b];                          //搜索与该边相邻的另一个平面
        face add;
        if(F[f].ok){
            if(dblcmp(P[p],F[f])> eps)
                //如果从 p 点能看到该面 f,则继续深度探索 f 的 3 条边,以更新新的凸面
                dfs(p,f);
            else{
                //如果从 p 点看不到 f 面,则 p 点和 a、b 点组成一个三角形
                add.a = b;
                add.b = a;
                add.c = p;
                add.ok = true;
                g[p][b] = g[a][p] = g[b][a] = num;
                F[num++] = add;
            }
        }
    }
    void dfs(int p,int now){                      //维护凸包,如果点 p 在凸包外则更新凸包
        F[now].ok = 0;
        deal(p,F[now].b,F[now].a);
        deal(p,F[now].c,F[now].b);
        deal(p,F[now].a,F[now].c);
    }
    bool same(int s,int t){                       //判断两个面是否为同一面
        Point3 &a = P[F[s].a];
        Point3 &b = P[F[s].b];
        Point3 &c = P[F[s].c];
        return fabs(volume4(a,b,c,P[F[t].a]))< eps &&
               fabs(volume4(a,b,c,P[F[t].b]))< eps &&
               fabs(volume4(a,b,c,P[F[t].c]))< eps;
    }
    //构建三维凸包
    void create(){
        int i,j,tmp;
        face add;
        num = 0;
        if(n < 4)return;
        //前 4 个点不共面
        bool flag = true;
        for(i = 1;i < n;i++){                     //使前两个点不共点
            if(Len(P[0] - P[i])> eps){
                swap(P[1],P[i]);
                flag = false;
                break;
            }
        }
        if(flag)return;
```

```
        flag = true;
        //使前 3 个点不共线
        for(i = 2;i < n;i++){
            if(Len(Cross(P[0] - P[1],P[1] - P[i]))> eps){
                swap(P[2],P[i]);
                flag = false;
                break;
            }
        }
        if(flag)return;
        flag = true;
        //使前 4 个点不共面
        for(int i = 3;i < n;i++){
            if(fabs(Dot(Cross(P[0] - P[1], P[1] - P[2]),P[0] - P[i]))> eps){
                swap(P[3],P[i]);
                flag = false;
                break;
            }
        }
        if(flag)return;
        for(i = 0;i < 4;i++){            //构建初始四面体(4 个点为 p[0]、p[1]、p[2]、p[3])
            add.a = (i + 1) % 4;
            add.b = (i + 2) % 4;
            add.c = (i + 3) % 4;
            add.ok = true;
            if(dblcmp(P[i],add)> 0)swap(add.b,add.c);
                //保证逆时针,即法向量朝外,这样新点才可看到
            g[add.a][add.b] = g[add.b][add.c] = g[add.c][add.a] = num;
                //逆向的有向边保存
            F[num++] = add;
        }
        for(i = 4;i < n;i++){            //构建更新凸包
            for(j = 0;j < num;j++){
              //判断点是否在当前三维凸包内,i 表示当前点,j 表示当前面
                if(F[j].ok&&dblcmp(P[i],F[j])> eps){
                        //对当前凸包面进行判断,看点能否看到这个面
                    dfs(i,j);            //点能看到当前面,更新凸包的面
                    break;
                }
            }
        }
        tmp = num;
        for(i = num = 0;i < tmp;i++)
          if(F[i].ok)
            F[num++] = F[i];
    }
    //凸包的表面积
    double area(){
        double res = 0;
        for(int i = 0;i < num;i++)
          res += Area2(P[F[i].a],P[F[i].b],P[F[i].c]);
```

```
        return res/2.0;
    }
    //体积
    double volume(){
        double res = 0;
        Point3 tmp(0,0,0);
        for(int i = 0;i < num;i++)
            res += volume4(tmp,P[F[i].a],P[F[i].b],P[F[i].c]);
        return fabs(res/6.0);
    }
    //表面三角形个数
    int triangle(){
        return num;
    }
    //表面多边形个数
    int polygon(){
        int i,j,res,flag;
        for(i = res = 0;i < num;i++){
            flag = 1;
            for(j = 0;j < i;j++)
              if(same(i,j)){
                  flag = 0;
                  break;
              }
            res += flag;
        }
        return res;
    }
};
CH3D hull;
int main(){
    while(scanf("%d",&hull.n) == 1){
        for(int i = 0;i < hull.n;i++)
            scanf("%lf%lf%lf",&hull.P[i].x,&hull.P[i].y,&hull.P[i].z);
        hull.create();
        printf("%d\n",hull.polygon());
    }
    return 0;
}
```

【习题】

hdu 4273,三维凸包重心。

hdu 3662。

11.4 几何模板


视频讲解

下面给出了本章的模板代码,以便于用户编程时参考。

```
const double pi = acos(-1.0);                     //高精度圆周率
const double eps = 1e-8;                          //偏差值
const int maxp = 1010;                            //点的数量
int sgn(double x){                                //判断 x 是否等于 0
    if(fabs(x) < eps)  return 0;
    else return x<0?-1:1;
}
int Dcmp(double x, double y){                     //比较两个浮点数：0 为相等；-1 为小于；1 为大于
    if(fabs(x - y) < eps) return 0;
    else return x<y ?-1:1;
}
//---------------平面几何：点和线--------------------------
struct Point{                                     //定义点及其基本运算
    double x,y;
    Point(){}
    Point(double x,double y):x(x),y(y){}
    Point operator + (Point B){return Point(x+B.x,y+B.y);}
    Point operator - (Point B){return Point(x-B.x,y-B.y);}
    Point operator * (double k){return Point(x*k,y*k);}    //长度增大 k 倍
    Point operator / (double k){return Point(x/k,y/k);}    //长度缩小 k 倍
    bool operator == (Point B){return sgn(x-B.x) == 0 && sgn(y-B.y) == 0;}
    bool operator < (Point B){                             //比较两个点,用于凸包计算
        return sgn(x-B.x)<0 || (sgn(x-B.x) == 0 && sgn(y-B.y)<0);}
};
typedef Point Vector;                                      //定义向量
double Dot(Vector A,Vector B){return A.x*B.x + A.y*B.y;}   //点积
double Len(Vector A){return sqrt(Dot(A,A));}               //向量的长度
double Len2(Vector A){return Dot(A,A);}                    //向量长度的平方
//A 与 B 的夹角
double Angle(Vector A,Vector B){return acos(Dot(A,B)/Len(A)/Len(B));}
double Cross(Vector A,Vector B){return A.x*B.y - A.y*B.x;} //叉积
//三角形 ABC 面积的两倍
double Area2(Point A, Point B, Point C){return Cross(B-A, C-A);}
//两点的距离,用两种方式实现
double Distance(Point A, Point B){return hypot(A.x-B.x,A.y-B.y);}
double Dist(Point A,Point B){
    return sqrt((A.x-B.x)*(A.x-B.x) + (A.y-B.y)*(A.y-B.y));}
//向量 A 的单位法向量
Vector Normal(Vector A){return Vector(-A.y/Len(A), A.x/Len(A));}
//向量是否平行或重合
bool Parallel(Vector A, Vector B){return sgn(Cross(A,B)) == 0;}
Vector Rotate(Vector A, double rad){                       //向量 A 逆时针旋转 rad 度
    return Vector(A.x*cos(rad) - A.y*sin(rad), A.x*sin(rad) + A.y*cos(rad));
}
struct Line{
    Point p1,p2;                                           //线上的两个点
    Line(){}
    Line(Point p1,Point p2):p1(p1),p2(p2){}
    //根据一个点和倾斜角 angle 确定直线,0<=angle<pi
    Line(Point p,double angle){
```

```
        p1 = p;
        if(sgn(angle - pi/2) == 0){p2 = (p1 + Point(0,1));}
        else{p2 = (p1 + Point(1,tan(angle)));}
    }
    //ax + by + c = 0
    Line(double a,double b,double c){
        if(sgn(a) == 0){
            p1 = Point(0, - c/b);
            p2 = Point(1, - c/b);
        }
        else if(sgn(b) == 0){
            p1 = Point( - c/a,0);
            p2 = Point( - c/a,1);
        }
        else{
            p1 = Point(0, - c/b);
            p2 = Point(1,( - c - a)/b);
        }
    }
};
typedef Line Segment;                    //定义线段,两端点是 Point p1,p2
//返回直线倾斜角,0<=angle<pi
double Line_angle(Line v){
    double k = atan2(v.p2.y - v.p1.y, v.p2.x - v.p1.x);
    if(sgn(k) < 0)k += pi;
    if(sgn(k - pi) == 0)k -= pi;
        return k;
}
//点和直线的关系:1 为在左侧;2 为点在右侧;0 为点在直线上
int Point_line_relation(Point p,Line v){
    int c = sgn(Cross(p - v.p1,v.p2 - v.p1));
    if(c < 0)return 1;                   //1: p 在 v 的左边
    if(c > 0)return 2;                   //2: p 在 v 的右边
    return 0;                            //0: p 在 v 上
}
//点和线段的关系: 0 为点 p 不在线段 v 上; 1 为点 p 在线段 v 上
bool Point_on_seg(Point p, Line v){
    return sgn(Cross(p - v.p1, v.p2 - v.p1)) == 0 &&
        sgn(Dot(p - v.p1,p - v.p2)) <= 0;
}
//两直线的关系:0 为平行,1 为重合,2 为相交
int Line_relation(Line v1, Line v2){
    if(sgn(Cross(v1.p2 - v1.p1,v2.p2 - v2.p1)) == 0){
        if(Point_line_relation(v1.p1,v2) == 0)  return 1;   //1: 重合
        else return 0;                                      //0: 平行
    }
    return 2;                                               //2: 相交
}
//点到直线的距离
double Dis_point_line(Point p, Line v){
    return fabs(Cross(p - v.p1,v.p2 - v.p1))/Distance(v.p1,v.p2);
```

```
}
//点在直线上的投影
Point Point_line_proj(Point p, Line v){
    double k = Dot(v.p2 - v.p1,p - v.p1)/Len2(v.p2 - v.p1);
    return v.p1 + (v.p2 - v.p1) * k;
}
//点 p 对直线 v 的对称点
Point Point_line_symmetry(Point p, Line v){
    Point q = Point_line_proj(p,v);
    return Point(2 * q.x - p.x,2 * q.y - p.y);
}
//点到线段的距离
double Dis_point_seg(Point p, Segment v){
    if(sgn(Dot(p - v.p1,v.p2 - v.p1))< 0 || sgn(Dot(p - v.p2,v.p1 - v.p2))< 0)
                                                    //点的投影不在线段上
        return min(Distance(p,v.p1),Distance(p,v.p2));
    return Dis_point_line(p,v);                     //点的投影在线段上
}
//求两直线 ab 和 cd 的交点,在调用前要保证两直线不平行或重合
Point Cross_point(Point a,Point b,Point c,Point d){      //Line1:ab,Line2:cd
    double s1 = Cross(b - a,c - a);
    double s2 = Cross(b - a,d - a);                     //叉积有正负
    return Point(c.x * s2 - d.x * s1,c.y * s2 - d.y * s1)/(s2 - s1);
}
//两线段是否相交: 1 为相交,0 为不相交
bool Cross_segment(Point a,Point b,Point c,Point d){    //Line1:ab,Line2:cd
    double c1 = Cross(b - a,c - a),c2 = Cross(b - a,d - a);
    double d1 = Cross(d - c,a - c),d2 = Cross(d - c,b - c);
    return sgn(c1) * sgn(c2)< 0 && sgn(d1) * sgn(d2)< 0;
                                                    //注意交点是端点的情况不算在内
}
//---------------平面几何: 多边形----------------
struct Polygon{
    int n;                                          //多边形的顶点数
    Point p[maxp];                                  //多边形的点
    Line v[maxp];                                   //多边形的边
};
//判断点和任意多边形的关系: 3 为点上; 2 为边上; 1 为内部; 0 为外部
int Point_in_polygon(Point pt,Point * p,int n){         //点 pt,多边形 Point * p
    for(int i = 0;i < n;i++){                       //点在多边形的顶点上
        if(p[i] == pt)return 3;
    }
    for(int i = 0;i < n;i++){                       //点在多边形的边上
        Line v = Line(p[i],p[(i + 1) % n]);
        if(Point_on_seg(pt,v)) return 2;
    }
    int num = 0;
    for(int i = 0;i < n;i++){
        int j = (i + 1) % n;
        int c = sgn(Cross(pt - p[j],p[i] - p[j]));
        int u = sgn(p[i].y - pt.y);
```

```
            int v = sgn(p[j].y - pt.y);
            if(c > 0 && u < 0 && v >= 0) num++;
            if(c < 0 && u >= 0 && v < 0) num--;
        }
        return num != 0;                       //1 为内部; 0 为外部
    }
    //多边形面积
    double Polygon_area(Point *p, int n){  //从原点开始划分三角形
        double area = 0;
        for(int i = 0;i < n;i++)
            area += Cross(p[i],p[(i + 1) % n]);
        return area/2;                         //面积有正负,不能简单地取绝对值
    }
    //求多边形的重心
    Point Polygon_center(Point *p, int n){
        Point ans(0,0);
        if(Polygon_area(p,n) == 0) return ans;
        for(int i = 0;i < n;i++)
            ans = ans + (p[i] + p[(i + 1) % n]) * Cross(p[i],p[(i + 1) % n]);
        return ans/Polygon_area(p,n)/6.;
    }
    //Convex_hull()求凸包.凸包顶点放在 ch 中,返回值是凸包的顶点数
    int Convex_hull(Point *p,int n,Point *ch){
        sort(p,p + n);                         //对点排序: 按 x 从小到大排序,如果 x 相同,按 y 排序
        n = unique(p,p + n) - p;               //去除重复点
        int v = 0;
        //求下凸包.如果 p[i]是右拐弯的,这个点不在凸包上,往回退
        for(int i = 0;i < n;i++){
            while(v > 1 && sgn(Cross(ch[v - 1] - ch[v - 2],p[i] - ch[v - 2]))<= 0)
                v--;
            ch[v++] = p[i];
        }
        int j = v;
        //求上凸包
        for(int i = n - 2;i >= 0;i--){
            while(v > j && sgn(Cross(ch[v - 1] - ch[v - 2],p[i] - ch[v - 2]))<= 0)
                v--;
            ch[v++] = p[i];
        }
        if(n > 1) v--;
        return v;                              //返回值 v 是凸包的顶点数
    }

    //---------------平面几何: 圆-----------------
    struct Circle{
        Point c;                               //圆心
        double r;                              //半径
        Circle(){}
        Circle(Point c,double r):c(c),r(r){}
        Circle(double x,double y,double _r){c = Point(x,y);r = _r;}
    };
```

```
//点和圆的关系: 0 为点在圆内, 1 为点在圆上, 2 为点在圆外
int Point_circle_relation(Point p, Circle C){
    double dst = Distance(p,C.c);
    if(sgn(dst - C.r) < 0) return 0;           //点在圆内
    if(sgn(dst - C.r) == 0) return 1;          //圆上
    return 2;                                   //圆外
}
//直线和圆的关系: 0 为直线在圆内, 1 为直线和圆相切, 2 为直线在圆外
int Line_circle_relation(Line v,Circle C){
    double dst = Dis_point_line(C.c,v);
    if(sgn(dst-C.r) < 0) return 0;             //直线在圆内
    if(sgn(dst-C.r) == 0) return 1;            //直线和圆相切
    return 2;                                   //直线在圆外
}
//线段和圆的关系: 0 为线段在圆内, 1 为线段和圆相切, 2 为线段在圆外
int Seg_circle_relation(Segment v,Circle C){
    double dst = Dis_point_seg(C.c,v);
    if(sgn(dst-C.r) < 0) return 0;             //线段在圆内
    if(sgn(dst-C.r) == 0) return 1;            //线段和圆相切
    return 2;                                   //线段在圆外
}
//直线和圆的交点.pa、pb 是交点.返回值是交点的个数
int Line_cross_circle(Line v,Circle C,Point &pa,Point &pb){
    if(Line_circle_relation(v, C) == 2) return 0;  //无交点
    Point q = Point_line_proj(C.c,v);          //圆心在直线上的投影点
    double d = Dis_point_line(C.c,v);          //圆心到直线的距离
    double k = sqrt(C.r*C.r-d*d);
    if(sgn(k) == 0){                            //一个交点,直线和圆相切
        pa = q;
        pb = q;
        return 1;
    }
    Point n=(v.p2-v.p1)/ Len(v.p2-v.p1);       //单位向量
    pa = q + n*k;
    pb = q - n*k;
    return 2;                                   //两个交点
}

//-------------------三维几何----------------
//三维: 点
struct Point3{
    double x,y,z;
    Point3(){}
    Point3(double x,double y,double z):x(x),y(y),z(z){}
    Point3 operator + (Point3 B){return Point3(x+B.x,y+B.y,z+B.z);}
    Point3 operator - (Point3 B){return Point3(x-B.x,y-B.y,z-B.z);}
    Point3 operator * (double k){return Point3(x*k,y*k,z*k);}
    Point3 operator / (double k){return Point3(x/k,y/k,z/k);}
    bool operator == (Point3 B){
        return sgn(x-B.x) == 0 && sgn(y-B.y) == 0 && sgn(z-B.z) == 0;}
};
```

```
typedef Point3 Vector3;
//点积.和二维点积函数同名.C++允许函数同名
double Dot(Vector3 A,Vector3 B){return A.x * B.x + A.y * B.y + A.z * B.z;}
//叉积
Vector3 Cross(Vector3 A,Vector3 B){
    return Point3(A.y * B.z - A.z * B.y,A.z * B.x - A.x * B.z,A.x * B.y - A.y * B.x);}
double Len(Vector3 A){return sqrt(Dot(A,A));}        //向量的长度
double Len2(Vector3 A){return Dot(A,A);}             //向量长度的平方
double Distance(Point3 A, Point3 B){                 //A、B 的距离
    return sqrt((A.x - B.x) * (A.x - B.x) +
                (A.y - B.y) * (A.y - B.y) + (A.z - B.z) * (A.z - B.z));
}
//A 与 B 的夹角
double Angle(Vector3 A,Vector3 B){return acos(Dot(A,B)/Len(A)/Len(B));}
//三维: 线
struct Line3{
    Point3 p1,p2;
    Line3(){}
    Line3(Point3 p1,Point3 p2):p1(p1),p2(p2){}
};
typedef Line3 Segment3;                              //定义线段,两端点是 Point p1,p2
//三维: 三角形面积的两倍
double Area2(Point3 A,Point3 B,Point3 C){return Len(Cross(B - A, C - A));}
//三维: 点到直线的距离
double Dis_point_line(Point3 p, Line3 v){
    return Len(Cross(v.p2 - v.p1,p - v.p1))/Distance(v.p1,v.p2);
}
//三维: 点在直线上
bool Point_line_relation(Point3 p,Line3 v){
    return sgn(Len(Cross(v.p1 - p,v.p2 - p))) == 0
                && sgn(Dot(v.p1 - p,v.p2 - p)) == 0;
}
//三维: 点到线段的距离
double Dis_point_seg(Point3 p, Segment3 v){
    if(sgn(Dot(p - v.p1,v.p2 - v.p1)) < 0 || sgn(Dot(p - v.p2,v.p1 - v.p2)) < 0)
        return min(Distance(p,v.p1),Distance(p,v.p2));
    return Dis_point_line(p,v);
}
//三维: 点 p 在直线上的投影
Point3 Point_line_proj(Point3 p, Line3 v){
    double k = Dot(v.p2 - v.p1,p - v.p1)/Len2(v.p2 - v.p1);
    return v.p1 + (v.p2 - v.p1) * k;
}
//三维: 平面
struct Plane{
    Point3 p1,p2,p3;                                 //平面上的 3 个点
    Plane(){}
    Plane(Point3 p1,Point3 p2,Point3 p3):p1(p1),p2(p2),p3(p3){}
};
//平面法向量
Point3 Pvec(Point3 A,Point3 B,Point3 C){ return Cross(B - A,C - A);}
```



```
Point3 Pvec(Plane f){return Cross(f.p2 - f.p1,f.p3 - f.p1);}
//四点共平面
bool Point_on_plane(Point3 A,Point3 B,Point3 C,Point3 D){
    return sgn(Dot(Pvec(A,B,C),D - A)) == 0;
}
//两平面平行
int Parallel(Plane f1, Plane f2){
    return Len(Cross(Pvec(f1),Pvec(f2))) < eps;
}
//两平面垂直
int Vertical (Plane f1, Plane f2){
    return sgn(Dot(Pvec(f1),Pvec(f2))) == 0;
}
//直线与平面的交点 p,返回值是交点的个数
int Line_cross_plane(Line3 u,Plane f,Point3 &p){
    Point3 v = Pvec(f);                          //平面的法向量
    double x = Dot(v, u.p2 - f.p1);
    double y = Dot(v, u.p1 - f.p1);
    double d = x - y;
    if(sgn(x) == 0 && sgn(y) == 0) return -1;    //-1: v 在 f 上
    if(sgn(d) == 0) return 0;                    //0: v 与 f 平行
    p = ((u.p1 * x) - (u.p2 * y))/d;             //1: v 与 f 相交
    return 1;
}
//四面体有向体积×6
double volume4(Point3 A,Point3 B,Point3 C,Point3 D){
    return Dot(Cross(B - A,C - A),D - A);}
```

11.5　小　结

几何题是竞赛初学者的难关,很多竞赛队甚至没有队员深入研究几何题,以至于在赛场上看到几何题直接放弃。

几何题往往逻辑烦琐,需要细致地编程,很考验编码能力。本章提到的内容,竞赛队的所有队员都应精通,并且指定其中一人深入研究,做到能轻松解决中等难度以上的题目。

第 12 章　ICPC 区域赛真题

前面各章节讲解了竞赛所需要的知识点。在参赛之前，队员有必要了解 ICPC 区域赛各赛区的总体情况，并通过解读 ICPC 区域赛真题做到心中有数，确定一个冲刺的目标。

历年的世界总决赛（World Finals）题目、区域赛题目都可以在官网 icpc. baylor. edu 浏览和提交①。其中，中国大陆往年的 ICPC 区域赛题目，acm. hdu. edu. cn 也有收录。

在一次区域赛中，为了区分参赛队员的能力，出题者需要考虑多方面的因素才能出一套合适的题目。大体上应该能考查竞赛队员的 5 种能力，包括编码、计算思维、逻辑推理、算法知识、团队合作。难度上的区分如表 12.1 所示。

表 12.1　难度上的区分

奖牌	编码	计算思维	逻辑推理	算法知识	团队合作	总分
铜牌	**	**	**	**	*	9
银牌	****	***	***	***	***	16
金牌	*****	*****	*****	*****	*****	25

从能力考查上看，铜牌 1 星或 2 星；银牌 3 星或 4 星；金牌 5 星。再考虑到铜牌、银牌、金牌的获奖比例按 3∶2∶1 逐渐缩小，可以得出结论：从铜牌提升到银牌比较容易，从银牌提升到金牌很难。

奖牌和参赛队 3 个队员的能力直接相关。如果有一人编码快，做题熟练，靠他一人就可以得到铜牌；如果 3 人都能进行大量训练，编码得心应手，算法知识大量掌握，加上一定的团队合作，可以得到银牌；在银牌的基础上，如果 3 人在平时训练中喜欢深入思考，不怕复杂的编码，把多种算法和数据结构融会贯通，团队合作紧密，再经历长期的练习，可以冲击金牌。

金牌、银牌队员是未来的 IT 精英，铜牌队员则需要继续努力。

12.1　ICPC 亚洲区域赛（中国大陆）情况

每年中国大陆赛区有 4～6 个，每个赛区的参赛队伍有 250 个左右，参赛学校 100 多个。奖牌设置比例是金牌 10%、银牌 20%、铜牌 30%，共 60%的参赛队得奖。例如，2015 年、2017 年亚洲区域赛中国大陆赛区相关信息见表 12.2 和表 12.3。

---

① ICPC live archive，网上简称 LA（https://icpcarchive. ecs. baylor. edu/）。

表 12.2　2015 年亚洲区域赛中国大陆赛区各站比赛情况统计

2015 年	长春	沈阳	合肥	北京	上海	EC-Final
题目数量	13	13	10	11	12	13
金牌题数	5～8	5～8	4～7	7～8	6～9	≥7
银牌题数	4～5	4～5	2～4	5～6	5～6	5～7
铜牌题数	4	3	1	4	4～5	3～5
参赛队数	220	172	97	200	199	288

表 12.3　2017 年亚洲区域赛中国大陆赛区各站比赛情况统计

2017 年	沈阳	西安	青岛	北京	南宁	乌鲁木齐	EC-Final
题目数量	13	11	11	10	11	10	13
金牌题数	6～10	6～10	3～6	5～9	8～11	7～10	9～11
银牌题数	5～6	4～6	3	3～5	6～7	5～7	6～9
铜牌题数	4～5	3～4	2～3	2～3	4～6	3～5	4～6
参赛队数	180	350	360	190	228	94	300

12.2　ICPC 区域赛题目解析



视频讲解

2015 年 11 月 22 日，亚洲区域赛上海站于华东理工大学举行，共有 120 所大学、199 队参加，来自中国大陆、中国香港(4 校 7 队)，以及朝鲜(1 校 1 队)、蒙古(1 校 3 队)等国家和地区，是一次典型的、有影响力的亚洲区域赛。

题目由电子科技大学退役金牌队员张鑫航、何云鹏拟定。题目难度分布合理，有很好的区分度[①]。

本次比赛共 12 道题目，题目在 ICPC 官网有存档[②]，或者在 hdu 上提交，题号是 5572～5584。读者在看下面的内容之前，为了更好地理解题目，提高自己的思考能力，最好先自己尝试做题。

铜牌：4 道或 5 道题，F、K、L、A、B 题。

银牌：5 道或 6 道题，加上 D 题。

金牌：6 道题以上，加上 E、G、I 题。

C、H、J 题有部分做出。

参赛队解题时间(AC 时间)如表 12.4 所示。

---

① 现场赛排名：https://perma.cc/VJ2A-B642。
官方档案：https://icpc.baylor.edu/regionals/finder/shanghai-2015/standings(短网址：t.cn/R397ena)。

② https://icpcarchive.ecs.baylor.edu/index.php?option=com_onlinejudge&Itemid=8&category=691(短网址：t.cn/R39zGy4)。

表 12.4　AC 时间(分钟,中位值)

题　　目	F	K	L	A	B	D	E	G	I
金牌(≥6 道)	7	25	48	101	97	239	229[1]	269	270
银牌(5 道或 6 道)	8	40	64	140	146	255	184	188	
铜牌(4 道或 5 道)	10	40	98	218	213				

下面按难易程度,由简到难对 9 道题目进行详细的讲解。

12.2.1　F 题 Friendship of Frog(hdu 5578)

Time Limit:2000/1000ms(Java/Others)

Memory Limit:65536/65536KB(Java/Others)[2]

Problem Description:

N 只青蛙站成一排,它们来自不同的国家。每个国家用一个小写字母表示。相邻青蛙的距离(例如第 1 个和第 2 个青蛙、第 $N-1$ 和第 N 个青蛙等)是 1。如果两只青蛙来自同一个国家,那么它们是朋友。

距离最小的一对朋友是最亲密的。帮忙找出这个距离是多少。

Input:

第 1 行是一个整数 T,表示测试用例的个数。

每一个测试用例只包括一串长度为 N 的字符串,其中第 i 个字符表示第 i 个青蛙的国籍。

Output:

对每个测试用例,需要输出"Case #x:y",其中 x 表示第几个用例,从 1 开始计数;y 是结果。如果没有来自同一个国家的青蛙,输出-1。

Limits:

$1 \leqslant T \leqslant 50$;

80%的数据,$1 \leqslant N \leqslant 100$;

100%的数据,$1 \leqslant N \leqslant 1000$;

字符串只包括小写字母。

Sample Input	Sample Output
2 abcecba abc	Case #1:2 Case #2:-1

【题解】

难度等级:极简单。本题是所谓的“签到题”,即参赛的所有队伍都能 AC 的简单题。

① 画线表示只有部分队伍做出来。

② Time Limit 和 Memory Limit 是本题的时间和空间限制,但是现场赛所发的题目一般不会给出,需要参赛队自己判断是否超时和超内存。这和在线判题的 OJ 不同,为方便学习,OJ 一般会给出这两个参数。Time Limit 和编程语言也有关系,Java 程序比 C、C++程序慢,所以 Java 的 Time Limit 更大一些,一般是 C、C++的 3 倍以上。

从读题到提交代码，参赛队 AC 的最短时间(First Blood，FB)是 3 分钟。铜牌队伍在 10 分钟左右 AC。

能力考核：编码能力。

本题逻辑简单，很容易理解，不涉及复杂的算法，代码也很短，参赛队员只要了解竞赛的入门知识就能做出来。

(1) 时间复杂度。在读题时，首先注意到数据长度 N 很小，$1 \leqslant N \leqslant 1000$，因此可以采用时间复杂度为 $O(N^3)$ 的算法。

(2) 逻辑和算法。由于题目很简单，首先想到暴力的方法。思路是从头开始检查第 i 个字符($0 \leqslant i \leqslant N$)，逐个比对它后面的 $N-i$ 个字符，寻找相同的。每次比对时把最短距离记录下来，直到全部结束。其复杂度是 $O(N^2)$，所以这个方法是可行的。

(3) 扩展。

虽然该题是很简单的题目，但是如果数据很大，例如 $N \leqslant 10^5$，那么上述的方法就会超时，此时需要更好的思路。

用暴力法结合“剪枝”的技巧可以处理。下面示例程序的时间复杂度是 $O(N)$。

```
#include <bits/stdc++.h>
using namespace std;
const int N = 100010;
char s[N];
void solve() {
    scanf("%s", s);
    int n = strlen(s);
    int ans = -1;
    for(int i = 0; i < n; ++i) {          //从头到尾检查字符串,i是当前位置
        for(int j = 1; j <= 26 && i - j >= 0; ++j) {
                //剪枝技巧：由于小写字符一共有 26 个,所以字符串中两个相
                //同字符的最小距离不会超过 26,只需要检查 i 前面的 26 个字符
            if(s[i] == s[i - j]) {
                //j 从 1 开始递增,即从距离 i 最近的字符开始检查,
                //如果有相同字符,就 break; 其他未检查的距离更大,不用继续检查
                if(ans == -1 || j < ans) {
                    ans = j;
                }
                break;
            }
        }
    }
    printf("%d\n", ans);
}
int main() {
    int t;
    scanf("%d", &t);
    for(int i = 1; i <= t; ++i) {
        printf("Case #%d: ", i);
        solve();
    }
    return 0;
}
```

12.2.2　K 题 Kingdom of Black and White(hdu 5583)

Problem Description：

黑白国有两种青蛙：黑青蛙和白青蛙。N 只青蛙站成一行，有些是黑的，有些是白的。计算青蛙们的合力，计算规则如下：把青蛙们分成最小的部分，每部分是连续的，只包含一种颜色的青蛙；合力是每部分长度的平方和。

现在来了一个罪恶的老巫婆，她告诉青蛙们，她要改变青蛙的颜色，最多改变一只。这样青蛙们的合力就变了。

青蛙们想知道，巫婆完成她的工作后，可能的最大合力是多少。

Input：

第 1 行是一个整数 T，表示测试用例的个数。

每个测试用例只包含一个字符串，长度为 N，只包含字符'0'(表示一只黑青蛙)和'1'(表示一只白青蛙)。

Output：

对每个测试用例，需要输出"Case #x:y"，其中 x 表示第几个用例，从 1 开始计数；y 是结果。

Limits：

$1 \leqslant T \leqslant 50$；

60%的数据，$1 \leqslant N \leqslant 1000$；

100%的数据，$1 \leqslant N \leqslant 10^5$；

字符串值包含 0 和 1。

Sample Input	Sample Output
2 000011 0101	Case #1：26 Case #2：10

【题解】

难度等级：简单题，用暴力法求解。FB 时间是 10 分钟。铜牌队伍在 40 分钟左右 AC。

能力考核：计算复杂度、编码能力。

本题逻辑虽然简单，但是也需要灵活处理；不涉及复杂的算法，但是需要了解计算的复杂度并避免落入陷阱；代码比较短，但是有一定的技巧。本题可以考查基本的计算思维和较好的编码能力。

(1) 逻辑。根据 Sample Input 和 Sample Output 理解题意，当输入 000011 时，青蛙的合力是 $4^2+2^2=20$。改变一只青蛙的颜色，例如改成 000001，合力变为 $5^2+1^2=26$。判断最大的合力是 26，并输出。

可能的最大合力，出现在有 $N=10^5$ 只青蛙的情况下，且所有青蛙是一种颜色，此时合力是 $N^2=10^{10}<2^{64}$，可以用 64 位的 long long 类型表示。

(2) 计算复杂度。在解题时，首先应该注意数据的规模，即 $1 \leqslant N \leqslant 10^5$，这说明不能使用时间复杂度大于 $O(N^2)$ 的算法。



算法竞赛的初学者，如果没有注意到这一限制，就会落入陷阱，用以下简单的暴力方法，结果是 TLE：每改变一只青蛙的颜色，就重新计算合力，计算量是 $O(N)$；从头到尾一共可以改变 N 次；总复杂度是 $O(N^2)$。

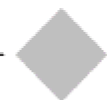
TLE 的错误代码

```
#include<bits/stdc++.h>
using namespace std;
const int N = 100010;
char s[N];
int n;
long long get_ans() {                      //计算合力,时间复杂度是 O(N)
    long long ans = 0;
    int cur = -1, len = 0;
    for(int i = 1; i<=n; ++i) {
        if(s[i] - '0' == cur)
            ++len;
        else {
            ans += 1LL * len * len;
            len = 1;
            cur = s[i] - '0';
        }
    }
    ans += 1LL * len * len;
    return ans;
}
void solve() {                             //总的时间复杂度是 O(N²),结果 TLE
    scanf("%s", s + 1);
    n = strlen(s + 1);
    long long ans = get_ans();
    for(int i = 1; i<=n; ++i) {            //逐个改变青蛙的颜色,可以改变 N 次
        s[i] = '1' + '0' - s[i];           //改变当前青蛙的颜色
        ans = max(ans, get_ans());         //计算合力并得到最大值
        s[i] = '1' + '0' - s[i];           //还原青蛙的颜色
    }
    printf("%lld\n", ans);
}
int main() {
    int t;
    scanf("%d", &t);
    for(int i = 1; i<=t; ++i) {
        printf("Case #%d: ", i);
        solve();
    }
    return 0;
}
```

(3) 优化和解决。在上述程序中，计算复杂度可以优化。每次改变一只青蛙的颜色后，因为只影响了相邻的青蛙序列，所以并不需要全部重新计算，只计算被影响的这部分就行了。这样，每改变一只青蛙的颜色，计算的时间复杂度差不多是 $O(1)$，总复杂度是 $O(N)$。

正确代码

```
#include <bits/stdc++.h>
using namespace std;
const int N = 100010;
#define square(x)  (x) * (x)
void solve() {
    int i, j = 1, k = 1;
    char s[N];
    int n;
    long long a[N] = {0};
    long long maxsum, oldsum = 0;
    scanf("%s", s + 1);
    n = strlen(s + 1);
    for(i = 2; i <= n; i++){              //把表示青蛙序列的"01"字符串改成数字,例如
        //把"000011001"改成数字 4221,并存放在数组 a[]中,处理起来更加便捷
      if(s[i] == s[i - 1])
         j++;
      else {
        a[k] = j;
        k++;
        j = 1;
      }
    }
    a[k] = j;
    for(i = 1; i <= k; i++){              //计算改变颜色前的合力
      oldsum = oldsum + a[i] * a[i];
    }
    maxsum = oldsum;
    for(i = 1; i <= k; i++){              //改变一只青蛙的颜色对合力的影响
                                          //只需要考虑以下两种情况:
      if(a[i] == 1)                       //如果长度是 1,说明这只青蛙是孤立的,
                                          //改变它的颜色后,可以和左右合并,
                                          //例如"00100"合并成"00000"
        maxsum = max(maxsum, oldsum + square(a[i - 1] + a[i] + a[i + 1]) - square(a[i - 1]) -
square(a[i]) - square(a[i + 1]));
      if(a[i]>= 2){                       //如果长度大于等于 2,可以分两次改变颜色:
           //改变最左边的,与左边的邻居合并,例如"0110"改成"0010";
           //改变最右边的,和右边的邻居合并,例如"0110"改成"0100"
           //如果长度大于等于 3,改变中间的,只会减小合力,
           //所以不用考虑,例如"01110"改成"01010",合力变小
        maxsum = max(maxsum, oldsum + square(a[i - 1] + 1) + square(a[i] - 1) - square(a[i -
1]) - square(a[i]));                      //给左边
        maxsum = max(maxsum, oldsum + square(a[i + 1] + 1) + square(a[i] - 1) - square(a[i +
1]) - square(a[i]));                      //给右边
      }
    }
    printf("%lld\n", maxsum);
}
int main() {
```



```
    int t;
    scanf(" % d", &t);
    for(int i = 1; i <= t; ++ i) {
        printf("Case # % d: ", i);
        solve();
    }
    return 0;
}
```

12.2.3　L 题 LCM Walk(hdu 5584)

Problem Description:

一只青蛙刚学会一些数论就迫不及待地想展示给女朋友看。

它坐在一个网格图上,行和列都是无限的。行的计数从底部开始,列也是这样。青蛙最初的位置是坐标(s_x,s_y),旅程开始了。

为了向女朋友炫耀它的数学天才,它使用了一种特别的跳跃方法。如果它在坐标(x,y)上,寻找一个可以被 x 和 y 都整除的最小的 z,然后向上或向右跳 z 步,下一步坐标可能是($x+z$,y)或(x,$y+z$)。

经过有限次跳跃后(可能是 0 步),它停在(e_x,e_y)处。然而它太累了,忘记了它的起始位置。

如果一个个去检查网格的所有坐标,那太笨了!请告诉青蛙一个聪明的办法,到达(e_x,e_y)的可能的起始位置有多少个?

Input:

第 1 行是一个整数 T,表示测试用例的个数。

每个测试用例包含两个整数 e_x、e_y,即目的地坐标。

Output:

对每个测试用例,需要输出"Case #x: y",其中 x 表示第几个用例,从 1 开始计数; y 是可能起点的个数。

Limits:

$1 \leqslant T \leqslant 1000$;

$1 \leqslant e_x, e_y \leqslant 10^9$。

Sample Input	Sample Output
3 6 10 6 8 2 8	Case #1: 1 Case #2: 2 Case #3: 3

【题解】

难度等级:简单题,数学。FB 时间是 18 分钟。铜牌队伍在 98 分钟左右 AC。

能力考核:数论中的最小公倍数和最大公约数问题、逻辑推理。

本题涉及了算法知识,不过比较容易,是简单的数论概念;推导过程需要有清晰、灵活

的推理；需要了解计算的复杂度；代码比较短。本题可以考查基本的算法知识和一定的逻辑推理能力。

(1) 算法复杂度。在解题时，首先应该注意数据的规模，即 $1 \leqslant e_x, e_y \leqslant 10^9$，这说明不能使用时间复杂度大于 $O(N)$ 的算法。

(2) 算法概念和逻辑推理。标题说明这是一个 LCM(最小公倍数)问题。

起点是(x,y)，终点是(e_x,e_y)，已知终点，反推起点。

z 是 x,y 的 LCM，假设 $x=pt, y=qt, z=pqt$。p 和 q 互质。

起点(x,y)，下一步可以走到两个位置$(x,y+z)$或$(x+z,y)$，下面分别讨论。

① 终点$(e_x,e_y)=(x,y+z)=(pt,qt+pqt)=(pt,q(1+p)t)$。

由于 p 和 $q(1+p)$互质，所以 t 是最大公约数，可得 $t=\text{GCD}(e_x,e_y)$。推导得到起点：

$$x=pt=e_x$$

$$y=qt=e_yt/(e_x+t)$$

这是一个可能的起点。

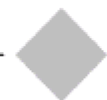
把起点当成新的终点，继续这个过程，直到结束。

需要注意，p、q、t 都是整数，在程序中需要判断。

② 终点$(e_x,e_y)=(x+z,y)$。

实际情况和①类似。注意到①中 $y+z>x$，即 $e_x<e_y$，②中 $e_x>e_y$。在编程时，只需要先按大小交换 e_x、e_y 的顺序，就可以合并成一种情况处理了。

```
#include <bits/stdc++.h>
int solve(long long ex, long long ey) {
  int ans = 1;
  long long t;
  while (true) {
     if (ex > ey)
    std::swap(ex, ey);
    t = std::__gcd(ex, ey);
              //p = ex / t;  q = ey/(ex + t); //在计算中 p 和 q 并未用到
    if ((ey % (ex + t)) == 0 ) {               //判断 q 是否为整数; t 和 p 肯定是整数,不用判断
        ey = ey * t/(ex + t);
        ans++;
    }
    else
        break;
  }
  return ans;
}
int main() {
    int T;
    long long ex, ey;
    scanf("%d", &T);
    for (int cas = 1; cas <= T; cas++) {
        scanf("%lld%lld", &ex, &ey);
        printf("Case #%d: %d\n", cas, solve(ex, ey));
    }
    return 0;
}
```



12.2.4　A 题 An Easy Physics Problem(hdu 5572)

Problem Description：

在一个无限光滑的桌面上有一个固定的大圆柱体，还有一个体积忽略不计的小球。

开始时，球静止于 A 点，给它一个初始速度和方向，如果球撞到圆柱体，它会弹回，没有能量损失。

经过一段时间，小球是否会经过 B 点？

Input：

第 1 行是一个整数 T，表示测试用例的个数。

每个测试用例有 3 行。

第 1 行有 3 个整数 Ox、Oy、r。圆柱体的中心是(Ox,Oy)，半径为 r。

第 2 行有 4 个整数 Ax、Ay、Vx、Vy。A 的坐标是(Ax,Ay)，初始方向矢量是(Vx,Vy)。

第 3 行有两个整数 Bx、By。B 的坐标是(Bx,By)。

Output：

对每个测试用例，需要输出"Case #x：y"，其中 x 表示第几个用例，从 1 开始计数；如果球会经过 B 点，y 是"Yes"，否则 y 是"No"。

Limits：

$1 \leqslant T \leqslant 100$；

$|\text{Ox}|, |\text{Oy}| \leqslant 1000$；

$1 \leqslant r \leqslant 100$；

$|\text{Ax}|, |\text{Ay}|, |\text{Bx}|, |\text{By}| \leqslant 1000$；

$|\text{Vx}|, |\text{Vy}| \leqslant 1000$；

$\text{Vx} \neq 0$ 或 $\text{Vy} \neq 0$；

A 和 B 都在圆柱体的外面，而且不重合。

Sample Input	Sample Output
2 0 0 1 2 2 0 1 −1 −1 0 0 1 −1 2 1 −1 1 2	Case #1：No Case #2：Yes

【题解】

难度等级：中等题，几何。FB 时间是 38 分钟。铜牌队伍在 220 分钟左右 AC。

能力考核：逻辑思维、较强的编码能力。

本题代码较长，逻辑较复杂，但是很容易理解，也不涉及复杂的算法。这种题是考查编码能力的典型题目，在编码时需要认真处理几何模板、逻辑关系等。

本题的代码已经在 11.2.1 节中作为例题详细给出。

12.2.5 B 题 Binary Tree(hdu 5573)

Problem Description:

青蛙国王住在一个无限长的树的根部。根据法律,每个结点应该连接下一层的两个结点,构成一个完整的二叉树。

因为国王的数学很牛,它为每个结点配置了一个数字。特别地,树的根就是国王住的地方,是 1,记为 froot=1。对每个结点 u,标签是 fu,左子结点是 fu×2,右子结点是 fu×2+1。国王对它的树王国很满意。

时间流逝,国王病了。根据黑魔法,如果国王能收集 N 个幽灵宝石,可以让它再活 N 年。开始时,国王位于根部,幽灵宝石的数量是 0,然后它往下走,每次往左子结点或右子结点走。在结点 x 处,这个结点的数字是 f_x(记住 froot=1),它可以选择把幽灵宝石增加 f_x 或者减少 f_x。它从根部开始走,访问 K 个 node(包括根结点),在每个结点处加或者减去上述的数字。如果数字最后是 N,它就成功了。注意幽灵宝石有魔法,幽灵宝石的数字 N 可能是负数。

给定 n、K,帮助国王收集 n 个幽灵宝石,不多不少访问 K 个结点。

Input:

第 1 行是一个整数 T,表示测试用例的个数。

每个测试用例包括两个整数 n 和 K,即国王要收集的幽灵宝石的数量、访问的结点数量。

Output:

对每个测试用例,先输出"Case #x:",其中 x 表示第几个用例,从 1 开始计数。

下面有 K 行,每一行'a b':a 是青蛙访问的结点标签;b 是'+'或'-',表示加或减 a。

保证至少有一个结果成立,如果有很多成立,可以输出任何一个。

Limits:

$1 \leqslant T \leqslant 100$;

$1 \leqslant n \leqslant 10^9$;

$n \leqslant 2^K \leqslant 2^{60}$。

Sample Input	Sample Output
2 5 3 10 4	Case #1: 1 + 3 - 7 + Case #2: 1 + 3 + 6 - 12 +

【题解】

难度等级：中等题，模拟题，构造。FB 时间是 44 分钟。铜牌队伍在 213 分钟左右 AC。

能力考核：计算思维、逻辑思维。

本题不需要复杂的算法，编码也不长，但是需要灵活处理，找到间接的解决方案。这是典型的考查思维能力的题目，具体是考查对二进制的领悟能力。这种思维能力需要聪明的头脑和长期的编程训练。

首先考虑计算复杂度。本题如果用暴力的方法，简单地罗列所有可能的走法是不行的。从第 1 层根结点走到第 K 层，一共有 2^{K-1} 个路径，由于每个结点上可取正负，那么每个路径上又有 2^K 种组合，合起来大概是 4^K，而 $K \leqslant 60$，肯定会 TLE。

类似这种题目，“灵机一动”非常重要。比如本题的思路是只要一直沿着最左边(加上第 K 层最左边的右子结点)走到第 K 层，就能找到一个答案。知道了这一点，后面就简单了。虽然题目中的限制条件 $n \leqslant 2^K$ 给出了暗示，但是想到这一点仍然很难。这也是为什么平时做题训练时尽量不要看题解，而是应该多自己思考，锻炼思维能力。如果靠看别人的题解知道这个方法，然后再去完成编码，收获是很小的。

上述思路证明如下：

(1) 二叉树最左边的那条边，从上到下相加，满足条件 $n \leqslant 2^K$。从第 1 层沿着最左边走到第 K 层(图 12.1 中路径是 1-2-4-8)，最大值是 $n = 2^0 + 2^1 + 2^2 + 2^3 + 2^4 + \cdots + 2^{K-1} = 2^K - 1$，如果在第 K 层选择最左边的右子结点(图 12.1 中路径是 1-2-4-9)，那么最大值是 $N = 2^K$。

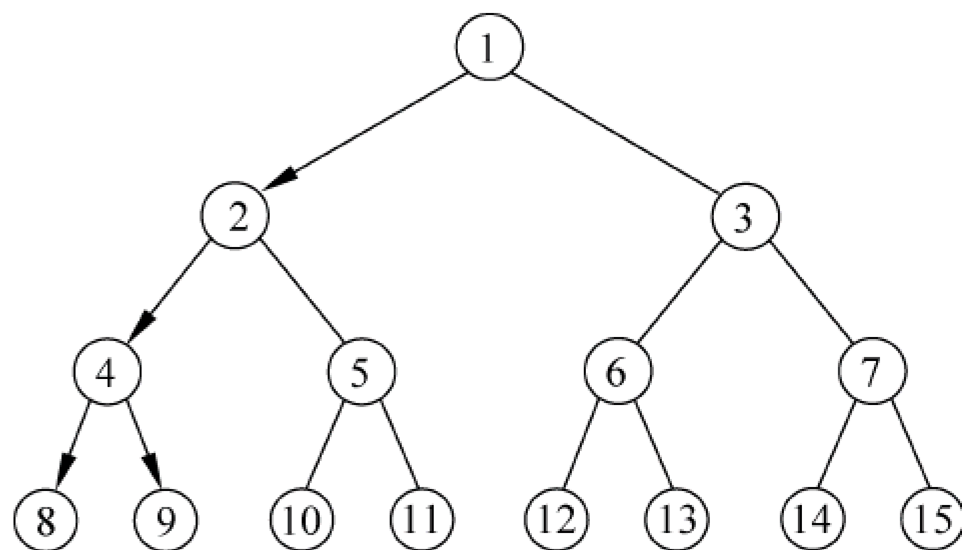


图 12.1　B 题

(2) 对于给定的 K，沿着最左边走(最后一层可以走右边)可以实现 $1 \leqslant n \leqslant 2^K$ 中的所有值，也就是对任何 n 都能找到一个答案。请自己证明。下面用一个例子来说明：令 $K=5$，最左边的数依次是 1,2,4,8,16。最大 $n=32$，现在证明 1～32 中所有的数都能用 1,2,4,8,16(最后一层可以走右边)的组合获得。实际上，一直走左边可以得到奇数，然后在最后一层走右边能得到偶数，所以只需要考虑奇数就行了。

$31=16+8+4+2+1$。联想到 31 的二进制表示：$31=11111_2$；用 11111 表示这条路径，其中 1 表示'+'，0 表示'-'。

$29=16+8+4+2-1$，$29=31-2$，把上面 $16+8+4+2+1$ 中的 $+1$ 变成 -1 即可。用 11110 表示这条路径。11110 可以这样计算得到：$31-(31-29)/2=30=11110_2$。

$27=16+8+4-2+1$，$27=31-4$，把 $+2$ 变成 -2，2 的二进制是 10_2。用 11101 表示这条路径，$31-(31-27)/2=29=11101_2$。

25＝16＋8＋4－2－1，25＝31－6，把＋3 变成－3，3 的二进制是 11_2。用 11100 表示这条路径，31－(31－25)/2＝28＝11100_2。

23＝16＋8－4＋2＋1，23＝31－8，把＋4 变成－4；用 11011 表示这条路径。

21＝16＋8－4＋2－1，21＝31－10，把＋5 变成－5；用 11010 表示这条路径。

……

每个奇数都能实现。

在编程时，结合二进制的特点，很容易写出程序。其复杂度为 $O(K)$。

```
#include <algorithm>
typedef long long LL;
int main(){
    int num, n, K, odd;
    scanf("%d", &num);
    for(int i = 1; i <= num; ++i){
        scanf("%d%d", &n, &K);
    if(n % 2)
            odd = 0;            //n是奇数,每一层都是最左边的数
        else {
            odd = 1;            //n是偶数,转换为比它小1的奇数处理。最后一层取右边的数
            n--;
        }
        LL pp = (LL)pow(2,K) - 1;
            //二进制数为全1的数。例如K=5时,pp=31,二进制是11111
        LL kk = pp - (pp - n)/2;   //kk的二进制表示,就是一个可行的路径。
                                   //其中为1的是'+',为0的是'-'。原因见上文的证明
        LL pos = 0;                //当前层数,从国王的顶层开始
        printf("Case #%d:\n", i);
        while(kk > 1) {            //不处理最后一层,最后一层有奇偶问题
          if(kk & 1)               //二进制数的个位数是当前的层
                                   //这个位置是'1',表示要加
            printf("%lld %c\n", (LL)pow(2,pos), '+');
        else                       //二进制数的个位数是'0',表示要减
            printf("%lld %c\n", (LL)pow(2,pos), '-');
        kk = kk >> 1;              //二进制数右移一次,把处理过的移走
                                   //新的个位数是下一层
        pos++;
      }
//下面处理最后一层。如果n是偶数,最后一层取右边
      if(kk & 1)
          printf("%lld %c\n", (LL)pow(2,pos) + odd, '+');
      else
          printf("%lld %c\n", (LL)pow(2,pos) + odd, '-');
    }
    return 0;
}
```

12.2.6 D 题 Discover Water Tank(hdu 5575)

Problem Description：

水箱里面住着很多青蛙，但是它们都不知道水箱里有多少水。

水箱的高度无限，但是底部狭窄。水箱底部长 N，宽只有 1。

$N-1$ 个板子把水箱隔成 N 部分，每部分底部大小是 1×1，板子的高度不同。水不能穿过板子，但是如果水平面比板子高，根据基本的物理规律，水会从板子上面漫过去。

青蛙国王想知道水箱的细节，它派人选择了 M 个点，看这些点上有没有水。

例如，每次它选择(x,y)，表示在水箱的第 x 部分$(1\leqslant x\leqslant N$，从左到右计数$)$，在高度$(y+0.5)$的地方检查是否有水。

国王得到了 M 个结果，但是它发现有些可能是错的。国王想知道正确结果的最大可能个数有多少。

Input：

第 1 行是一个整数 T，表示测试用例的个数。

每个测试用例的第 1 行是两个数 N 和 M，即水箱隔成 N 部分、测量 M 次。

每个测试用例的第 2 行包括 $N-1$ 个整数，即 $h_1,h_2,\cdots,h_{N-1}$，h_i 表示第 i 个板子的高度。

下面有 M 行，第 i 行的格式为'x y z'，表示测量结果。如果第 x 个水箱高$(y+0.5)$处没有水，那么 $z=0$，否则 $z=1$。

Output：

对每个测试用例，需要输出"Case #x：y"，其中 x 表示第几个用例，从 1 开始计数；y 是正确结果的最大可能数字。

Limits：

$1\leqslant T\leqslant100$；

90%的数据，$1\leqslant N\leqslant1000$，$1\leqslant M\leqslant2000$；

100%的数据，$1\leqslant N\leqslant10^5$，$1\leqslant M\leqslant2\times10^5$；

$1\leqslant h_i\leqslant10^9$，$1\leqslant i\leqslant N-1$；

对每个结果，$1\leqslant x\leqslant N$，$1\leqslant y\leqslant10^9$，$z$ 是 0 或 1。

Sample Input	Sample Output
2 3 4 3 4 1 3 1 2 1 0 2 2 0 3 3 1 2 2 2 1 2 0 1 2 1	Case #1：3 Case #2：1

【题解】

难度等级：难题，综合。FB 时间是 119 分钟。金牌队伍在 240 分钟左右 AC，银牌有少

数队伍做出。

能力考核：高级数据结构(左偏树)、STL库、逻辑思维、编码能力。

本题有复杂的逻辑，大规模的数据，需要结合数据结构、算法、STL库，是典型的难题，综合考查逻辑、算法、编码等多方面的能力。

(1) 计算复杂度。读题时首先应注意到本题较大的数据规模，即板子的数量 $1 \leqslant N \leqslant 10^5$，以及测量的个数 $1 \leqslant M \leqslant 2 \times 10^5$，因此计算复杂度比 $O(NM)$ 小。

(2) 理解题目。

图12.2是第1个测试样例。图中'×'表示无水，即 $z=0$；'O'表示有水，即 $z=1$。在这个例子中，最大的正确数字是3，即第1个水箱的测量结果是错的，后面3个结果都是对的。

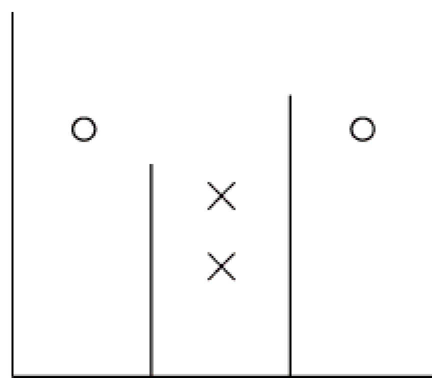


图12.2　D题

本题的解决思路是比较清晰的，即从低到高，逐渐给水箱加水，然后计算正确的测量个数。步骤如下：

① 从假设水箱没有水开始，此时'×'的个数就是答案，记为ans。

② 逐一检查每个'O'的有水记录，即给这个水箱加水，直到这个记录的高度。可以想到，按从低到高的顺序检查所有的'O'，逻辑上是正确的，而且比较简单。

③ 上一步中对每个'O'的检查，加水到'O'的高度后，它可能向左、向右溢出。检查被它溢出的水箱，在当前水位高度下有多少'×'？有多少'O'？'O'的数量减去'×'的数量，差值为 d，更新ans，ans=ans+d。

④ 检查完所有'O'，输出ans。

在以上思路中，③是最关键的。检查每个'O'向左、向右的溢出，复杂度是 $O(N)$；一共有 M 个'O'，总复杂度是 $O(MN)$。而 $1 \leqslant N \leqslant 10^5$，$1 \leqslant M \leqslant 2 \times 10^5$，$O(MN)$ 很大，显然不能用暴力的方法去检查每个'O'的溢出。那么怎么做呢？因为是按从低到高的顺序检查'O'，在检查更高的'O'时，前面检查过的、相邻水箱的、较低的'O'的检查结果可以直接拿来用。或者说，一些相邻的水箱可以合并为一个大水箱。

当水向左、向右溢出以后，被溢出到的小水箱合并成一个大水箱，这样 $O(M)$ 次枚举'O'的复杂度就是 $O(N)$，而不是 $O(MN)$ 了。

在合并的时候，不仅要合并当前水位之下的'O'和'×'的数量，还要维护大水箱的左、右两边是哪些水箱以及是哪些挡板。'O'的数量可以看当前枚举了多少O，'×'的数量可以用一个能表示顺序的数据结构来维护，这个数据结构维护结构体 (x, y) 表示'×'在坐标为 (x, y) 的位置。在计算第 x 个小水箱的水位 h 下的'×'的时候，可以在删除数据结构最小元素的同时计数，直到 $y \geqslant h$（即最低位置的'×'在水位之上）为止。维护顺序的数据结构可以用堆或者平衡树，但是又需要数据能够快速地合并，因此使用由左偏树实现的可并堆。

概括起来，解题过程是以 $z=0$ 的测量总数为初始答案，通过枚举 $z=1$ 的情况来更新答案，维护水箱并处理水箱的合并，用左偏树实现的可并堆来处理水箱的合并及计算单次枚举的答案。总的复杂度是 $O(N\log N + N\log M)$。

下面是出题人提供的代码。

```
#include <bits/stdc++.h>
using namespace std;
const int N = 200100;
```

```
const int INF = 2000000000;
int n, m, h[N], st[N], wh[N], height[N * 2], val[N * 2][2], dp[N * 2][2];
int lca[N * 2][18], dep[N * 2];
vector<int> pos[2], off[2], high[N * 2][2];
vector<int> edge[2 * N];
void clear(){
    for(int i = 0; i <= n; ++ i) {
        edge[i].clear();
        high[i][0].clear(); high[i][1].clear();
    }
    pos[0].clear(); off[0].clear();
    pos[1].clear(); off[1].clear();
}
void pre_dfs(int u, int fa, int dist) {
    lca[u][0] = fa; dep[u] = dist;
    for(int i = 1; i < 18; ++ i)
        lca[u][i] = lca[lca[u][i - 1]][i - 1];
    for(int i = 0; i < edge[u].size(); ++ i) {
        int v = edge[u][i];
        pre_dfs(v, u, dist + 1);
    }
}
int get_node(int x, int y) {
    for(int i = 17; i >= 0; -- i)
        if(dep[x] > (1 << i) && height[lca[x][i]] <= y)
            x = lca[x][i];
    return x;
}
void add_edge(int x, int y) {
    edge[x].push_back(y);
}
void build_tree() {
    int tot = n, top = 0;
    h[n] = INF;height[0] = INF;
    for(int i = 1; i <= n; ++ i) {
        height[i] = 0; val[i][0] = val[i][1] = 0;
        wh[top] = i;
        while(top > 0 && st[top - 1] < h[i]) {
            ++ tot;
            height[tot] = st[top - 1];
            val[tot][0] = val[tot][1] = 0;
            int tid = top - 1;
            add_edge(tot, wh[top]);
            while(top > 0 && st[top - 1] == st[tid]){
                -- top;
                add_edge(tot, wh[top]);
            }
            wh[top] = tot;
        }
        st[top ++] = h[i];
    }
```

```
    n = tot;
    for(int i = 0; i < 18; ++ i)
        lca[0][i] = 0;
    pre_dfs(n, 0, 1);
    for(int i = 0; i < 2; ++ i) {
        for(int j = 0; j < pos[i].size(); ++ j) {
            int id = get_node(pos[i][j], off[i][j]);
            val[id][i] ++; high[id][i].push_back(off[i][j]);
        }
    }
}
void dfs(int u) {
    dp[u][0] = val[u][0];
    dp[u][1] = val[u][1];
    sort(high[u][0].begin(), high[u][0].end());
    sort(high[u][1].begin(), high[u][1].end());
    int p = 0, ret = 0;
    for(int i = 0; i < val[u][0]; ++ i) {
        while(p < val[u][1] && high[u][1][p] < high[u][0][i])
            ++ p;
        ret = max(ret, p + val[u][0] - i);
    }
    int flag = val[u][0], Mind = INF;
    for(int i = 0; i < edge[u].size(); ++ i) {
        int v = edge[u][i];
        dfs(v);
        dp[u][1] += dp[v][1];
        dp[u][0] += max(dp[v][0], dp[v][1]);
    }
    dp[u][0] = max(dp[u][0], dp[u][1] - val[u][1] + ret);
}
void solve() {
    scanf("%d%d", &n, &m);
    for(int i = 1; i < n; ++ i)
        scanf("%d", h + i);
    for(int i = 1; i <= m; ++ i) {
        int x, y, z;
        scanf("%d%d%d", &x, &y, &z);
        pos[z].push_back(x); off[z].push_back(y);
    }
    build_tree();
    dfs(n);
    printf("%d\n", max(dp[n][0], dp[n][1]));
    clear();
}
int main() {
    int t;
    scanf("%d", &t);
    for(int i = 1; i <= t; ++ i) {
        printf("Case #%d: ", i);
        solve();
```

```
    }
    return 0;
}
```

12.2.7　E 题 Expection of String(hdu 5576)

Problem Description：

青蛙刚学会了乘法，现在它想做一些练习。

它在纸上写了一个字符串，只包括数字和一个'×'符号。如果'×'出现在字符串前面或后面，它认为结果是0，否则它将按正常乘法来计算。

在练习之后，它想到一个新问题：对一个初始字符串，每次随机选两个字符交换位置，它交换了一次又一次，例如 K 次，它想知道新字符串计算结果的期望值。

可以知道所有交换的方法一共有 $\binom{n}{2}^K$ 种(即$(C_n^2)^K$)。如果期望的结果是 x，需要输出整数 $x\times\binom{n}{2}^K$。

Input：

第1行是一个整数 T，表示测试用例的个数。

每个测试用例的第1行是一个数 K，表示青蛙交换字符的次数。

每个测试用例的第2行是青蛙操作的字符串，只包括数字和一个乘法操作符'*'。

Output：

对每个测试用例，需要先输出"Case #x：y"，其中 x 表示第几个用例，从1开始计数；y 是结果。

由于 y 可能很大，用 10^9+7 取模。

Limits：

$1\leqslant T\leqslant 100$；

字符串长度为 L；

70%的数据，$1\leqslant L\leqslant 10$，$0\leqslant K\leqslant 5$；

95%的数据，$1\leqslant L\leqslant 20$，$0\leqslant K\leqslant 20$；

100%的数据，$1\leqslant L\leqslant 50$，$0\leqslant K\leqslant 50$。

Sample Input	Sample Output
2 1 1*2 2 1*2	Case #1：2 Case #2：6

【题解】

难度等级：难题。FB 时间是118分钟。部分金牌队伍做出，AC 时间在230分钟左右。

能力考核：DP、逻辑思维、编码能力。

本题的逻辑很复杂，数据大，是典型的难题，综合考查逻辑、算法、编码等多方面的能力。

题目的要求是一个字符串由数字和'×'号组成，每次操作可以交换其中任意两个符号(包括'×')，问 K 次操作后所有可能结果的和。

样例1，字符串"1 * 2"，交换一次，结果有$(C_n^2)^K=(C_3^2)^1=3$ 种情况，即 * 12、2 * 1、12 * 。期望值 $x=\frac{0}{3}+\frac{2}{3}+\frac{0}{3}=\frac{2}{3}$，输出 $x\times(C_n^2)^K=\frac{2}{3}\times3=2$。

样例2，字符串"1 * 2"，交换两次，结果可能有$(C_n^2)^K=(C_3^2)^2=9$ 种，分别是 1 * 2、21 * 、* 21、* 21、1 * 2、21 * 、21 * 、* 21、1 * 2，如图 12.3 所示。期望值 $x=\frac{2}{9}+\frac{2}{9}+\frac{2}{9}=\frac{6}{9}$，输出 $x\times(C_n^2)^K=\frac{6}{9}\times9=6$。

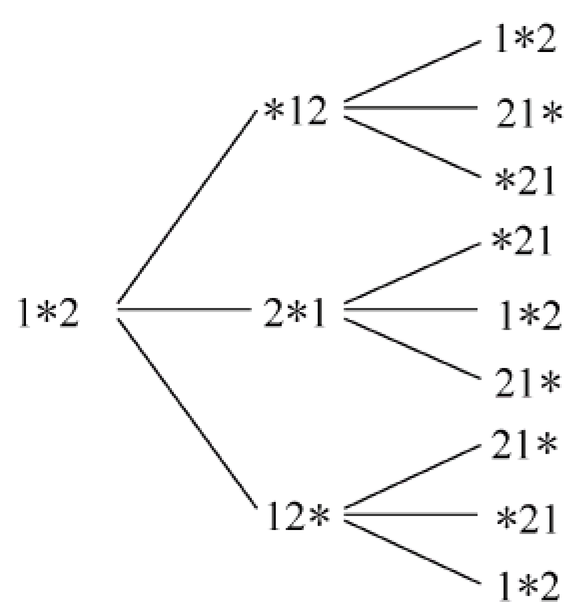


图 12.3　E 题

本题如果用暴力的方法，逐一检查每个结果，可能有$(C_n^2)^K\leqslant(C_{50}^2)^{50}$种情况，数据太大，不可能进行计算。

用 DP 实现，关键是递推式，复杂度为 $O(Kn^3)$。

下面是出题人提供的代码。

```
#include <bits/stdc++.h>
using namespace std;
#define ll long long
int K,n;
const ll mod = 1000000007;
ll dp[2][55][55][55];
ll ten[55],tot[55],tot_2[55][55];
string str;
int main(){
    int T;
    int cas = 1;
    cin >> T;
    ten[0] = ten[1] = 1;
    for (int i = 2; i <= 50; i++)
        ten[i] = ten[i - 1] * 10 % mod;
    while (T--) {
        cin >> K >> str;
        n = str.size();
        int now = 0;
        for (int i = 0; i < n; i++)
        if (str[i] == '*') {
            for (int j = 0; j < n; j++)
                for (int k = j + 1; k < n; k++)
                if (str[j] == '*' || str[k] == '*')
                    dp[now][i][j][k] = 0;
                else
                    dp[now][i][j][k] = (str[j] - '0') * (str[k] - '0');
        }
        else {
            for (int j = 0; j < n; j++)
```

```
            for (int k = 0; k < n; k++)
                dp[now][i][j][k] = 0;
    }
    ll lamb;
    if (n <= 3) lamb = 1;
    else lamb = (n - 3) * (n - 4) / 2 + 1;
    for (int iter = 0; iter < K; iter++) {
        now = 1 - now;
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                for (int k = 0; k < n; k++)
                    dp[now][i][j][k] = 0;
        for (int j = 0; j < n; j++)
            for (int k = j + 1; k < n; k++) {
                tot_2[j][k] = 0;
                for (int i = 0; i < n; i++)
                if (i != j && i != k)
                    tot_2[j][k] += dp[1 - now][i][j][k];
                tot_2[j][k] % = mod;
            }
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++)
            if (i != j) {
                tot[j] = 0;
                for (int k = 0; k < n; k++)
                if (k != i && k != j)
                    tot[j] += dp[1 - now][i][min(j,k)][max(j,k)];
                tot[j] % = mod;
            }
            for (int j = 0; j < n; j++)
                for (int k = j + 1; k < n; k++)
                if (j != i && k != i) {
                    dp[now][i][j][k] += dp[1 - now][i][j][k] * lamb;
                    dp[now][i][j][k] += tot[j] - dp[1 - now][i][j][k] + mod;
                    dp[now][i][j][k] += tot[k] - dp[1 - now][i][j][k] + mod;
                    dp[now][i][j][k] += tot_2[j][k] - dp[1 - now][i][j][k] + mod;
                    dp[now][i][j][k] += dp[1 - now][j][min(i,k)][max(i,k)]
                                      + dp[1 - now][k][min(i,j)][max(i,j)];
                }
        }
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                for (int k = 0; k < n; k++) {
                    dp[now][i][j][k] % = mod;
                }
    }
    ll ans = 0;
    for (int i = 1; i < n - 1; i++)
        for (int j = 0; j < i; j++)
            for (int k = i + 1; k < n; k++)
                ans += dp[now][i][j][k] * ten[i - j] % mod * ten[n - k] % mod;
```

```
        printf("Case # %d: %lld\n", cas++, ans % mod);
    }
}
```

12.2.8　G 题 Game of Arrays(hdu 5579)

Problem Description:

Tweek 和 Craig 是好朋友,总在一起玩。在做数学作业的时候,他们发明了一种新游戏。

首先,他们写了 3 个数组 A、B、C,每个有 N 个数字。接着,他们在黑板上写下:

$$A+B=C$$

如果等式满足,说明从 1 到 N 的所有位置 $A_i+B_i=C_i$ 都成立。

当然,开始的时候等式不是都成立。

很幸运,数组 A、B、C 的一些数字可以改变,一些不能改。那些可以改的数字的位置在游戏前固定好了。

在游戏中,Tweek 先走,然后两人轮流进行,每次可以改变一个数字。

每一次,游戏者可以从一个数组中选一个可改变的数字,减去 1。但是,不能出现负数,所以被选中的数字在做减法前不能是 0。

Tweek 的目标是在游戏中使等式成立,而 Craig 的目标是阻止。

当等式成立时游戏结束,Tweek 获胜。或者不存在可能的改变,$A+B\neq C$(至少有一个 $i\in[1,N]$,使得 $A_i+B_i\neq C_i$),Craig 获胜。

给定 A、B、C,以及每个数组可改变数字的位置。本题的任务是确定谁获胜。

Input:

第 1 行是一个整数 T,表示测试用例的个数。

每个测试用例的第 1 行是一个整数 K,表示数组 A、B、C 的长度。

每个测试用例的第 2 行和第 3 行描述数组 A。第 2 行包括 N 个整数 $A_1,A_2,\cdots,A_N$,表示数组 A 的元素。第 3 行包括 N 个整数 $u_1,u_2,\cdots,u_N$,如果 A_i 可改,u_i 是 1,否则 u_i 是 0。

每个测试用例的第 4 行和第 5 行描述数组 B。第 4 行包括 N 个整数 $B_1,B_2,\cdots,B_N$,表示数组 B 的元素。第 5 行包括 N 个整数 $v_1,v_2,\cdots,v_N$,如果 B_i 可改,v_i 是 1,否则 v_i 是 0。

每个测试用例的第 6 行和第 7 行描述数组 C。第 6 行包括 N 个整数 $C_1,C_2,\cdots,C_N$,表示数组 C 的元素。第 7 行包括 N 个整数 $w_1,w_2,\cdots,w_N$,如果 C_i 可改,w_i 是 1,否则 w_i 是 0。

Output:

对每个测试用例,需要先输出"Case #x: y",其中 x 表示第几个用例,从 1 开始计数;y 是获胜者。

Limits:

$1\leqslant T\leqslant 2000$;

75%的数据,$1\leqslant N\leqslant 10$;

95%的数据,$1\leqslant N\leqslant 50$;



100％的数据，$1 \leqslant N \leqslant 100$；
$0 \leqslant A_i, B_i, C_i \leqslant 10^9$；
u_i、v_i、w_i 值为 0 或 1。

Sample Input	Sample Output
3 2 4 3 1 1 4 4 0 1 5 5 0 0 2 4 4 1 1 4 4 0 0 5 5 0 0 2 4 4 1 1 4 4 0 0 4 4 0 0	Case #1: Tweek Case #2: Craig Case #3: Tweek

【题解】

难度等级：本题是“难题”。FB 时间是 188 分钟。部分金牌队伍做出，AC 时间在 270 分钟左右。

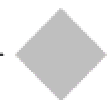
能力考核：数学、逻辑思维、编码能力。

本题综合考查逻辑、算法、编码等多方面的能力。

下面是出题人提供的代码。

```
#include<bits/stdc++.h>
using namespace std;
const int N = 1100;
int a[N], b[N], c[N], ta[N], tb[N], tc[N], n, d[N], add[N], del[N];
bool check() {
    int tot = 0;
    for(int i = 1; i<=n; ++i) {
        d[i] = a[i] + b[i] - c[i];
        add[i] = a[i] * ta[i] + b[i] * tb[i];
        del[i] = c[i] * tc[i];
```

```
            tot += abs(d[i]);
        }
        for(int i = 1; i <= n; ++ i) {
            if((d[i] > 0 && d[i] > add[i] - del[i]) ||
                    (d[i] < 0 && - d[i] > del[i] - add[i]))
                return 0;
            if((d[i] >= 0 && d[i] < add[i] - del[i]) ||
                    (d[i] <= 0 && - d[i] < del[i] - add[i])) {
                if(tot - abs(d[i]) > abs(d[i]))
                    return 0;
            }
        }
        return 1;
    }
    void solve(int cas) {
        scanf("%d", &n);
        for(int i = 1; i <= n; ++ i) scanf("%d", a + i);
        for(int i = 1; i <= n; ++ i) scanf("%d", ta + i);
        for(int i = 1; i <= n; ++ i) scanf("%d", b + i);
        for(int i = 1; i <= n; ++ i) scanf("%d", tb + i);
        for(int i = 1; i <= n; ++ i) scanf("%d", c + i);
        for(int i = 1; i <= n; ++ i) scanf("%d", tc + i);
        int win = 1;
        for(int i = 1; i <= n; ++ i) {
            if(a[i] + b[i] != c[i]) {
                win = 0;
                break;
            }
        }
        if(win){puts("Tweek"); return;}
        for(int i = 1; i <= n; ++i){
            if(ta[i] && a[i] > 0){
                a[i] --;
                if(check()){puts("Tweek");return;}
                a[i] ++;
            }
            if(tb[i] && b[i] > 0) {
                b[i] --;
                if(check()){puts("Tweek");return;}
                b[i] ++;
            }
            if(tc[i] && c[i] > 0) {
                c[i] --;
                if(check()){puts("Tweek");return;}
                c[i] ++;
            }
        }
        puts("Craig");
```



```
        return;
    }
    int main(){
        int t;
        scanf("%d", &t);
        for(int i = 1; i <= t; ++ i) {
            printf("Case #%d: ", i);
            solve(i);
        }
        return 0;
    }
```

12.2.9 I题 Infinity Point Sets(hdu 5581)

Problem Description：

这个故事来自一位古老的青蛙哲学家写的古书。

什么时候会是世界的尽头？也许最好的理解方法是使用几何进行计算。

起初，应该在一张纸上画几个点。每次选择两个点并用线段连接起来。当有两个线段交叉时，在交叉点上产生一个新点，将该点添加到纸上，并尝试像之前一样将其与前面的点连接。

应该一次又一次地执行此操作，继续绘制线段，并在可能的情况下添加点，直到没有新线段为止。然后是世界的尽头，旧的青蛙会死亡，新的时代将开始。

如你所见，不同的初始点导致不同的结果。对于一些点集合，世界的末日永远不会到来，我们称之为无穷大集合。

现在给 N 个点，在这 N 个点的所有可能子集(不包括空集，所以总共将有 2^N-1 个集)中有多少个不是无穷大？

Input：

第1行是一个整数 T，表示测试用例的个数。

每个测试用例都以整数 N 开始，它表示点的数量。

在下面的 N 行中，第 i 行包含两个整数 x_i 和 y_i，表示第 i 点的坐标(x_i, y_i)。

Output：

对每个测试用例，需要输出“Case #x：y”，其中 x 表示第几个用例，从1开始计数；y 是结果。

由于 y 可能很大，用 10^9+7 取模。

Limits：

$1 \leqslant T \leqslant 10$；

90%的数据，$1 \leqslant N \leqslant 100$；

100%的数据，$1 \leqslant N \leqslant 1000$；

$1 \leqslant x_i, y_i \leqslant 10^4$；

没有一对具有相同坐标的点。

Sample Input	Sample Output
2 4 0 0 0 2 2 2 2 0 5 0 0 0 2 2 2 2 0 1 2	Case #1：15 Case #2：30

【题解】

难度等级：本题是“难题”。FB 时间是 197 分钟。部分金牌队伍做出。

能力考核：几何、逻辑思维、编码能力。

本题综合考查逻辑、算法、编码等多方面的能力。题目大意是给出二维空间里 n 个点的坐标，求有多少个不同的子点集不是无限点集。无限点集的定义是将点集中的点两两相连，将线段产生的交点再加入点集中，继续上面的操作，如果操作能够无限地进行下去，则称之为无限点集。

图 12.4 所示的 4 种情况不是无限点集。

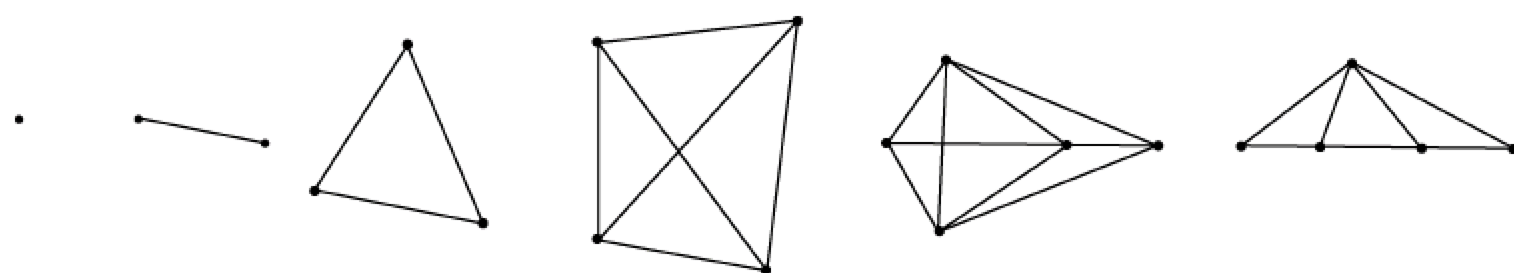


图 12.4　I 题

(1) 任意 1、2、3、4 个点。

(2) 3 点以上共线＋两侧各一点。生成的新的黑色的点也在线段上，操作不会无限地进行。

(3) 4 点以上共线＋任意一点。这种情况不会产生交点，操作也不会无限进行。

(4) 5 点及以上共线。

无限点集的情况例如图 12.5 所示。5 个外面的点，交点为 5 个内部的点，这样的操作能够无限地进行下去。

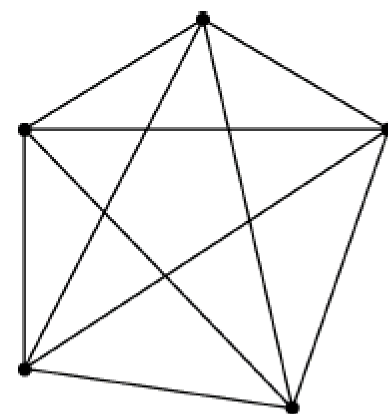


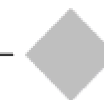
图 12.5　5 个点的情况

下面是出题人提供的代码。

```
#include <bits/stdc++.h>
typedef long long LL;
using namespace std;
const int V = 1100;
```

```
const int N = 1000;
const int P = 1000000007;
int rev[V], pt[V], C[V][V];
int Pow(int x, int y){
    int ret = 1;
    while(y){
        if(y & 1) ret =  (LL) ret * x % P;
        x = (LL) x * x % P;
        y /= 2;
    }
    return ret;
}
void init(){
    for(int i = 1; i <= N; ++i)
        rev[i] = Pow(i, P - 2);
    pt[0] = 1;
    for(int i = 1; i <= N; ++i)
        pt[i] = pt[i - 1] * 2 % P;
    memset(C, 0, sizeof(C));
    for(int i = 0; i <= N; ++i){
        C[i][0] = C[i][i] = 1;
        for(int j = 1; j < i; ++j)
            C[i][j] = (C[i - 1][j - 1] + C[i - 1][j]) % P;
    }
}
struct Point{
    int x, y;
}p[V];
struct PNode{
    int x, y, rev;
}Node[V];
bool EQ(PNode x, PNode y){
    if(x.x == y.x && x.x == 0) return true;
    if(x.x * y.x < 0) return false;
    return x.x * y.y == x.y * y.x;
}
bool Nodecmp(PNode x, PNode y){
    if(x.x * y.x <= 0) return x.x > y.x;
    if(x.x * y.y != x.y * y.x){
        if(x.x >= 0) return x.x * y.y > x.y * y.x;
        else return x.x * y.y > x.y * y.x;
    }
    return x.rev < y.rev;
}
int _, n;
/* 分为几部分: (1) 5 点及以上共线; (2) 任意 1、2、3、4 个点;
                (3) 4 点以上共线 + 任意一点; (4) 3 点以上共线 + 两侧各一点 */
int sol_line(int ln, int rn, int revn, int nown, int total){
    int ans = 0;
    for(int i = 4; i <= ln + rn; ++i)
        ans = (ans + (LL)C[ln + rn][i] * rev[i + 1] % P) % P;
```

```
    for(int i = 3; i <= ln + rn; ++i)
        ans = (ans + (LL)C[ln + rn][i] * rev[i + 1] % P * (n - ln - rn - 1) % P) % P;
    int D = revn - nown;
    int A = revn - D - rn;
    int c = total - A - ln - rn;
    int CD = c + D;
    int AB = n - 1 - ln - rn - CD;
    for(int i = 2; i <= ln + rn; ++i)
        ans = (ans + (LL)C[ln + rn][i] * rev[i + 1] % P * AB % P * CD % P) % P;
    return ans;
}
int mid_way[V];
int sol(){
    int ret = 0;
    for(int i = 1; i <= 4; ++i)
    ret = (ret + C[n][i]) % P;
    for(int i = 0; i < n; ++i){
        int revn = 0;
        for(int j = 0; j < n; ++j){
            Node[j].x = p[j].x - p[i].x;
            Node[j].y = p[j].y - p[i].y;
            if(Node[j].y < 0 || (Node[j].y == 0 && Node[j].x < 0)){
                Node[j].x = -Node[j].x;
                Node[j].y = -Node[j].y;
                Node[j].rev = 1;
                ++revn;
            }
            else Node[j].rev = -1;
        }
        sort(Node, Node + n, Nodecmp);
        int ln = 0, rn = 0, midn = 0, nown = 0, total = 0, pre = -1;
        for(int j = 0; j < n; ++j){
            if(Node[j].x == 0 && Node[j].y == 0) continue;
            if(pre != -1 && !EQ(Node[j], Node[pre])){
                ret += sol_line(ln, rn, revn, nown, total);
                ret % = P;
                mid_way[midn++] = (LL) ln * rn % P;
                ln = rn = 0;
            }
            if(Node[j].rev == -1)  ++ln;
            else ++nown, ++rn;
            ++total;
            pre = j;
        }
        mid_way[midn++] = (LL) ln * rn % P;
        ret += sol_line(ln, rn, revn, nown, total);
        ret % = P;
        int mids = 0;
        for(int j = 0; j < midn; ++j) mids = (mids + mid_way[j]) % P;
        for(int j = 0; j < midn; ++j){
            ret = (ret - (LL)(mids - mid_way[j]) * mid_way[j] % P * rev[2] % P) % P;
```



```
            if(ret < 0) ret += P;
        }
    }
    return ret;
}
int main(){
    init();
    scanf("%d", &_);
    for(int ca = 1; ca <= _; ++ca){
        scanf("%d", &n);
        for(int i = 0; i < n; ++i)
        scanf("%d%d", &p[i].x, &p[i].y);
        printf("Case #%d: %d\n", ca, sol());
    }
    return 0;
}
```

参 考 文 献

[1] 刘汝佳,陈锋.算法竞赛入门经典训练指南[M].北京:清华大学出版社,2012.
[2] 刘汝佳.算法竞赛入门经典[M].2版.北京:清华大学出版社,2014.
[3] 余立功.ACM/ICPC算法训练教程[M].北京:清华大学出版社,2013.
[4] 秋叶拓哉,岩田阳一,北川宜稔.挑战程序设计竞赛[M].巫泽俊,等译.2版.北京:人民邮电出版社,2013.
[5] 金博,郭立,于瑞云.计算几何及应用[M].哈尔滨:哈尔滨工业大学出版社,2012.
[6] 俞勇.ACM国际大学生程序设计竞赛算法与实现[M].北京:清华大学出版社,2013.
[7] Levitin A.算法设计与分析基础[M].潘彦,译.2版.北京:清华大学出版社,2007.
[8] Cormen T H,Leiserson C E.算法导论[M].潘金贵,等译.北京:机械工业出版社,2006.



“十二五”普通高等教育本科国家级规划教材



21世纪大学本科计算机专业系列教材

在线教学版　教学资源　练习与测试　互动教学　智能学习

蒋本珊　编著

计算机组成原理（第4版）

http://www.tup.com.cn

- 根据教育部“高等学校计算机科学与技术专业规范”组织编写
- 与美国 ACM 和 IEEE CS *Computing Curricula* 最新进展同步
- 本书第1版和第2版先后被评为北京高等教育精品教材
- 本书第2版和第3版被评为普通高等教育“十一五”国家级规划教材和“十二五”普通高等教育本科国家级规划教材

清华大学出版社



"十二五"普通高等教育本科国家级规划教材
北京高等教育精品教材
21 世纪大学本科计算机专业系列教材

计算机组成原理

（第 4 版）

蒋本珊　编著

清华大学出版社
北　京

内 容 简 介

本书系统地介绍了计算机的基本组成原理和内部工作机制。主要内容分成两个部分，由9章组成：第1、2章介绍计算机的基础知识；第3～9章介绍计算机的各子系统（包括运算器、存储器、控制器、总线、外部设备和输入输出子系统等）的基本组成原理、设计方法、相互关系以及各子系统互相连接构成整机系统的技术。

本书讲述了计算机的基本原理和基本概念，并注意与实用性和先进性相结合。全书内容由浅入深，通俗易懂，每章之后均附有习题，便于自学。

本书可以作为高等院校计算机及相关专业“计算机组成原理”课程的教材，也可供从事计算机工作的工程技术人员参考。

图书在版编目（CIP）数据

计算机组成原理/蒋本珊编著. —4版. —北京：清华大学出版社，2019
（21世纪大学本科计算机专业系列教材）
ISBN 978-7-302-53021-3

Ⅰ. ①计…　Ⅱ. ①蒋…　Ⅲ. ①计算机组成原理－高等学校－教材　Ⅳ. ①TP301

中国版本图书馆CIP数据核字(2019)第093911号

责任编辑：张瑞庆
封面设计：常雪影
责任校对：梁　毅
责任印制：沈　露

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦A座　　**邮　　编：**100084
社 总 机：010-62770175　　**邮　　购：**010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn，010-62795954
印 装 者：北京嘉实印刷有限公司
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　张：**22.25　　**字　数：**536千字
版　　次：2004年3月第1版　2019年8月第4版　　**印　次：**2019年8月第1次印刷
定　　价：56.90元

产品编号：083950-01

第 4 版前言

FOREWORD

《计算机组成原理》一书最初写作于 2003 年，先后出版了第 1～3 版，累计发行 20 万册。本书的第 3 版出版于 2013 年 5 月，它与配套的参考书《计算机组成原理学习指导与习题解析》（第 3 版）和《计算机组成原理教师用书》（第 3 版）均受到了读者的欢迎和好评，印刷 13 次。由于第 3 版出版迄今已 6 年有余，应广大同行和读者的强烈要求，从 2018 年 9 月底开始了本次修订。

本次修订未对全书的框架结构进行调整，只是对部分内容进行了必要的调整、更新、删减，并且参考国内外相关资料增加了一些最新的概念和技术。

修订后的本书保持了原书概念清楚、通俗易懂的风格，在强调基本原理、基本概念的同时，也注意了实用性和先进性。

本书每章后附有大量的习题，在与本书完全配套的《计算机组成原理教师用书》（第 4 版）第一章的“教材习题解答”版块中给出了本书中全部习题的解答，以供读者参考。

本人所公布的电子邮箱（bs.jiang@163.com）多年来收到很多同行和读者的来信，均已逐一认真回复。对于一些好的建议，在本次修订中也有所体现。希望修订之后的本书能对读者更有帮助，欢迎继续来信提出宝贵意见和建议。

作　者

2019 年 3 月

第3版前言

FOREWORD

《计算机组成原理》一书是本人积多年教学经验和体会精心写作完成的，本书第1版出版至今已近10年。承蒙专家和读者的厚爱，收获了不少成果。例如，《计算机组成原理》2004年被评为北京市精品教材，2007年入选教育部普通高等教育“十一五”国家级规划教材；《计算机组成原理》（第2版）2009年获得兵工高校优秀教材一等奖，《计算机组成原理》（第2版）以及配套参考书《计算机组成原理学习指导与习题解析》（第2版）、《计算机组成原理教师用书》（第2版）共3本书在2011年被评为北京市精品教材，《计算机组成原理》（第2版）2012年入选教育部“十二五”普通高等教育本科国家级规划教材。

本次修订有比较大的调整。首先是增加了总线一章，使全书的总章数由8章变为9章。然后，对原有的各章也都进行了必要的调整、更新、删减，补充了不少新的内容，特别是为适应计算机飞速发展的需要，参考国内外相关资料，增加了不少最新的概念和技术，仅补充和更新的内容就近4万字。

修订后的本书保持了原书概念清楚、通俗易懂的风格，在强调基本原理、基本概念的同时，也注意了实用性和先进性。

此次修订得到欧阳凌、潘海军的大力支持和帮助，他们在资料收集、图表制作、书稿整理等方面参与了工作，在此表示感谢。

继续欢迎读者来信提出意见和建议，电子邮箱：bs.jiang@163.com。对于读者的来信，本人将会逐一回复。

作　者

2013年5月

第 2 版前言

FOREWORD

承蒙读者的厚爱，本书第 1 版出版仅三年，已连续印刷 10 次，总印数达到 41 000 册。本书 2004 年被评为北京市精品教材，2007 年入选教育部普通高等教育“十一五”国家级规划教材。本书作为主教材，与 2005 年以来相继出版的《计算机组成原理学习指导与习题解析》和《计算机组成原理教师用书》一起形成了一个比较完整的教材教学体系，可以适应大多数高校的计算机及相关专业“计算机组成原理”课程教学的需要，受到了许多老师的欢迎。

随着计算机技术发展的日新月异，对第 1 版教材中部分内容的更新也被提上了议事日程。此次修订，保留了原书的框架和风格，全书章节基本保持不变，但与原书相比，进行了必要的调整、删减，补充了一些新的内容，增加了不少新的概念和新的技术。

第 1 章增加哈佛结构的内容，第 3 章增加指令系统的发展的内容，第 5 章增加 DDR2 SDRAM 和 DDR3 SDRAM 等内容，第 6 章增加微处理器中的新技术的内容，第 7 章增加 SATA 硬盘、NCQ 技术和显卡等内容，第 8 章增加 PCI-Express 总线的内容。

除了这些大的变化，其余章节的内容也有一些修改和补充。

修订后的本书力求概念清楚、通俗易懂，并注意处理好基本原理、基本概念与实用性、先进性之间的关系。

本书第 1 版自面市以来，收到了许多同行和读者发来的电子邮件，对于读者的来信，本人均给予了一一回复和解答。希望修订之后的本书能对读者有所帮助，欢迎来信提出意见和建议。电子邮箱：bs.jiang@163.com。

作　者

2008 年 6 月

第1版前言

FOREWORD

“计算机组成原理”是计算机类各专业学生的必修核心课程之一，主要讨论计算机各大部件的基本组成原理，各大部件互连构成整机系统的技术。本课程在计算机学科中处于承上启下的地位，先修课程应包括计算机基础、数字电路等。本课程的参考教学时数为56～72学时。

全书共分8章：第1、2章介绍计算机的基础知识（概论、数据的机器层次表示），第3～8章介绍计算机的各子系统（指令系统、数值的机器运算、存储系统和结构、中央处理器、外部设备、输入输出系统）的基本组成原理。

本书的内容与教学时数允许的分量相比偏多一些，有些内容对某些专业来说可能已在先导课程中讨论过，因此，在使用本教材时，可根据各专业的具体情况在章节上有选择地进行取舍。

计算机组成原理类的教材在国内已有不少，本书在下列几个方面具有一定的特色。

第一，本书是中国计算机学会和清华大学出版社共同规划的面向全国高等学校计算机专业本科生的“21世纪大学本科计算机专业系列教材”之一，内容覆盖了《中国计算机科学与技术学科教程2002(CCC2002)》对本课程所列出的知识单元。

第二，本书既强调计算机的基本概念和基础知识，对计算机的各大基本部件的组成原理、设计方法及相互关系都进行了较详细的描述，又注意与实际应用相结合，具有一定的针对性，以避免理论和实际脱节。

第三，计算机技术的发展日新月异，作为一本专业基础课教材，不可能也没有必要处处体现先进技术；有时过于求新，反倒会使内容显得高深且难以理解。本书在阐述中注意由浅入深、循序渐进，在讲清基本原理的基础上，再提出先进技术和新的发展方向，以降低学习的难度。

第四，本书根据各章节内容的要求按横向方式组织课程实例，而不拘泥于某一种具体的机型，以减少局限性，扩大读者的视野和适用面。考虑到目前国内的实际情况，实例以微、小型计算机为主。

第五，为了帮助读者建立整机概念，本书中介绍了一个仅有十几条指令的模型机，试图通过解剖这样一个小小的“麻雀”来介绍控制器的设计方法。

第六，各章内容相对独立，由浅入深，同时注意章节间内容的衔接，适合自学。

总之，本书力求做到内容全面、概念清楚、通俗易懂，并注意到实用性和先进性。

本书每章后附有大量的习题，为读者提供较多的练习机会。

本书还附有配套的电子教案,以便于教学使用,需要的教师可直接与清华大学出版社联系索取。

在本书编写过程中得到了“21世纪大学本科计算机专业系列教材”编委会的多次指导和建议。重庆大学计算机系袁开榜教授亲自仔细审阅了本书的全部内容,提出了许多宝贵的修改意见。清华大学出版社的编辑们也为本书的出版做了许多工作。在此对他们辛勤的工作和热情的支持表示诚挚的感谢!

由于时间的原因以及个人的水平限制,书上难免出现错误和不妥之处,欢迎同行和广大读者批评指正。如有问题可直接与作者联系,电子邮箱是:bs.jiang@163.com。

作　者

2003年11月

目 录

CONTENTS

第 1 章　概论 …… 1

1.1　电子计算机与存储程序控制 …… 1

1.1.1　电子计算机的发展 …… 1

1.1.2　存储程序概念 …… 2

1.2　计算机的硬件组成 …… 3

1.2.1　计算机的主要部件 …… 3

1.2.2　计算机各大部件之间连接 …… 4

1.2.3　不同对象观察到的计算机硬件系统 …… 5

1.2.4　冯・诺依曼结构和哈佛结构的存储器设计思想 …… 7

1.3　计算机系统 …… 9

1.3.1　硬件与软件的关系 …… 9

1.3.2　系列机和软件兼容 …… 9

1.3.3　计算机系统的多层次结构 …… 10

1.3.4　实际机器和虚拟机器 …… 11

1.4　计算机的工作过程和主要性能指标 …… 11

1.4.1　计算机的工作过程 …… 11

1.4.2　计算机的主要性能指标 …… 12

习题 …… 14

第 2 章　数据的机器层次表示 …… 15

2.1　数值数据的表示 …… 15

2.1.1　计算机中的数值数据 …… 15

2.1.2　无符号数和带符号数 …… 15

2.1.3　原码表示法 …… 16

2.1.4　补码表示法 …… 17

2.1.5　反码表示法 …… 18

2.1.6　3 种机器数的比较与转换 …… 19

2.2　机器数的定点表示与浮点表示 …… 20

2.2.1　定点表示法 …… 20

2.2.2 浮点表示法 …… 22
2.2.3 浮点数阶码的移码表示法 …… 24
2.2.4 浮点数尾数的基数 …… 25
2.2.5 IEEE 754 标准浮点数 …… 26
2.2.6 定点、浮点表示法与定点、浮点计算机 …… 28
2.3 非数值数据的表示 …… 29
2.3.1 字符和字符串的表示 …… 29
2.3.2 汉字的表示 …… 30
2.3.3 统一代码 …… 33
2.4 十进制数和数串的表示 …… 34
2.4.1 十进制数的编码 …… 34
2.4.2 十进制数串 …… 35
2.5 不同类型的数据表示举例 …… 36
2.5.1 C语言中的数据表示 …… 36
2.5.2 现代微型计算机系统中的数据表示 …… 38
2.6 数据校验码 …… 39
2.6.1 奇偶校验码 …… 40
2.6.2 汉明校验码 …… 42
2.6.3 循环冗余校验码 …… 45
习题 …… 46

第3章 指令系统 …… 49

3.1 指令格式 …… 49
3.1.1 机器指令的基本格式 …… 49
3.1.2 地址码结构 …… 50
3.1.3 指令的操作码 …… 52
3.2 寻址技术 …… 54
3.2.1 编址方式 …… 54
3.2.2 指令寻址和数据寻址 …… 55
3.2.3 基本的数据寻址方式 …… 56
3.2.4 变型或组合寻址方式 …… 61
3.3 堆栈与堆栈操作 …… 63
3.3.1 堆栈结构 …… 63
3.3.2 堆栈操作 …… 65
3.4 指令类型 …… 65
3.4.1 数据传送类指令 …… 65
3.4.2 运算类指令 …… 66
3.4.3 程序控制类指令 …… 67
3.4.4 输入输出类指令 …… 69

3.4.5　80x86 指令系统举例 …… 70
3.5　指令系统的发展 …… 72
3.5.1　x86 架构的扩展指令集 …… 72
3.5.2　从复杂指令系统到精简指令系统 …… 74
3.5.3　VLIW 和 EPIC …… 74
习题 …… 76

第 4 章　数值的机器运算 …… 79

4.1　基本算术运算的实现 …… 79
4.1.1　加法器 …… 79
4.1.2　进位的产生和传递 …… 80
4.1.3　并行加法器的快速进位 …… 81
4.2　定点加减运算 …… 83
4.2.1　原码加减运算 …… 83
4.2.2　补码加减运算 …… 84
4.2.3　补码的溢出判断与检测方法 …… 86
4.2.4　补码定点加减运算的实现 …… 88
4.3　带符号数的移位和舍入操作 …… 89
4.3.1　带符号数的移位操作 …… 89
4.3.2　带符号数的舍入操作 …… 90
4.4　定点乘法运算 …… 91
4.4.1　原码一位乘法 …… 91
4.4.2　补码一位乘法 …… 93
4.4.3　补码两位乘法 …… 96
4.4.4　阵列乘法器 …… 98
4.5　定点除法运算 …… 99
4.5.1　原码除法运算 …… 99
4.5.2　补码除法运算 …… 102
4.5.3　阵列除法器 …… 105
4.6　规格化浮点运算 …… 106
4.6.1　浮点加减运算 …… 106
4.6.2　浮点乘除运算 …… 108
4.6.3　浮点运算器的实现 …… 110
4.7　十进制整数的加法运算 …… 110
4.7.1　一位十进制加法运算 …… 110
4.7.2　十进制加法器 …… 112
4.7.3　多位十进制加法 …… 113
4.8　逻辑运算与实现 …… 114
4.9　运算器的基本组成与实例 …… 115

4.9.1 运算器结构 …… 115
4.9.2 ALU举例 …… 117
4.9.3 浮点运算器举例 …… 119
习题 …… 121

第5章 存储系统和结构 …… 123

5.1 存储系统的组成 …… 123
5.1.1 存储器分类 …… 123
5.1.2 存储系统层次结构 …… 125
5.2 主存储器的组织 …… 126
5.2.1 主存储器的基本结构 …… 126
5.2.2 主存储器的存储单元 …… 127
5.2.3 主存储器的主要技术指标 …… 128
5.2.4 数据在主存中的存放 …… 129
5.3 半导体随机存储器和只读存储器 …… 130
5.3.1 RAM记忆单元电路 …… 130
5.3.2 动态RAM的刷新 …… 133
5.3.3 RAM芯片分析 …… 135
5.3.4 半导体只读存储器 …… 138
5.3.5 半导体存储器的封装 …… 139
5.4 主存储器的连接与控制 …… 140
5.4.1 主存容量的扩展 …… 140
5.4.2 存储芯片的地址分配和片选 …… 143
5.4.3 主存储器和CPU的连接 …… 145
5.4.4 主存的校验 …… 146
5.4.5 PC系列微型计算机的存储器接口 …… 147
5.5 提高主存读写速度的技术 …… 150
5.5.1 主存与CPU速度的匹配 …… 150
5.5.2 FPM DRAM …… 151
5.5.3 EDO DRAM …… 151
5.5.4 SDRAM …… 152
5.5.5 DDR SDRAM …… 152
5.5.6 DDR2、DDR3、DDR4和DDR5 SDRAM …… 153
5.5.7 Rambus DRAM …… 154
5.5.8 多通道内存技术 …… 155
5.6 多体交叉存储技术 …… 156
5.6.1 并行访问存储器 …… 156
5.6.2 交叉访问存储器 …… 157
5.7 高速缓冲存储器 …… 158

5.7.1　高速缓存工作原理 …… 158
5.7.2　Cache 的读写操作 …… 159
5.7.3　地址映像 …… 159
5.7.4　替换算法 …… 161
5.7.5　更新策略 …… 161
5.7.6　微机中 Cache 技术的实现 …… 162
5.8　虚拟存储器 …… 163
5.8.1　虚拟存储器的基本概念 …… 163
5.8.2　页式虚拟存储器 …… 164
5.8.3　段式虚拟存储器 …… 164
5.8.4　段页式虚拟存储器 …… 165
5.8.5　快表与慢表 …… 166
习题 …… 166

第 6 章　中央处理器 …… 169

6.1　中央处理器的功能和组成 …… 169
6.1.1　CPU 的功能 …… 169
6.1.2　CPU 中的主要寄存器 …… 169
6.1.3　CPU 的组成 …… 171
6.1.4　CPU 的主要技术参数 …… 172
6.2　控制器的组成和实现方法 …… 176
6.2.1　控制器的基本组成 …… 176
6.2.2　控制器的硬件实现方法 …… 177
6.3　时序系统与控制方式 …… 178
6.3.1　时序系统 …… 178
6.3.2　控制方式 …… 180
6.3.3　指令运行的基本过程 …… 181
6.3.4　指令的微操作序列 …… 183
6.4　微程序控制原理 …… 184
6.4.1　微程序控制的基本概念 …… 185
6.4.2　微指令编码法 …… 186
6.4.3　微程序控制器的组成和工作过程 …… 188
6.4.4　微程序入口地址的形成 …… 189
6.4.5　后继微地址的形成 …… 190
6.4.6　微程序设计 …… 191
6.5　控制单元的设计 …… 193
6.5.1　简单的 CPU 模型 …… 193
6.5.2　组合逻辑控制单元设计 …… 196
6.5.3　微程序控制单元设计 …… 200

6.6 流水线技术 …… 202
6.6.1 重叠控制 …… 203
6.6.2 先行控制原理 …… 204
6.6.3 流水工作原理 …… 205
6.7 精简指令系统计算机 …… 207
6.7.1 RISC的特点和优势 …… 207
6.7.2 RISC基本技术 …… 209
6.8 微处理器中的新技术 …… 209
6.8.1 超标量和超流水线技术 …… 209
6.8.2 EPIC的指令级并行处理 …… 210
6.8.3 超线程技术 …… 210
6.8.4 双核与多核技术 …… 211
6.8.5 睿频加速技术 …… 213
习题 …… 214

第7章 总线 …… 217

7.1 总线概述 …… 217
7.1.1 总线的基本概念 …… 217
7.1.2 总线的分类 …… 218
7.1.3 总线的组成及性能指标 …… 219
7.2 总线仲裁 …… 221
7.2.1 集中仲裁方式 …… 221
7.2.2 分布仲裁方式 …… 223
7.3 总线定时控制 …… 223
7.3.1 同步定时方式 …… 223
7.3.2 异步定时方式 …… 224
7.4 总线标准 …… 224
7.4.1 系统总线标准 …… 225
7.4.2 外部总线标准 …… 227
习题 …… 230

第8章 外部设备 …… 231

8.1 外部设备概述 …… 231
8.1.1 外部设备的分类 …… 231
8.1.2 外部设备的地位和作用 …… 232
8.2 磁介质存储器的性能和原理 …… 233
8.2.1 磁介质存储器的读写 …… 233
8.2.2 磁介质存储器的技术指标 …… 234
8.2.3 数字磁记录方式 …… 235

8.2.4　编码方式的比较 …… 238
8.3　磁介质存储设备 …… 239
8.3.1　硬盘存储器的基本结构与分类 …… 239
8.3.2　硬盘驱动器 …… 239
8.3.3　硬盘的信息分布和磁盘地址 …… 240
8.3.4　硬盘存储器的技术参数 …… 241
8.3.5　硬盘的分区域记录 …… 244
8.3.6　硬盘的 NCQ 技术 …… 245
8.4　磁盘阵列 …… 245
8.4.1　RAID 简介 …… 245
8.4.2　RAID 的分级 …… 246
8.5　光盘存储器 …… 247
8.5.1　光盘存储器的类型 …… 248
8.5.2　光盘存储器的组成及工作原理 …… 248
8.5.3　光盘驱动器 …… 252
8.6　新型辅助存储器 …… 253
8.6.1　基于磁或磁光介质的可移动存储器 …… 253
8.6.2　基于电子器件的存储器 …… 255
8.7　键盘输入设备 …… 258
8.7.1　键开关与键盘类型 …… 258
8.7.2　键盘扫描 …… 258
8.7.3　微型计算机键盘 …… 260
8.8　其他输入设备 …… 262
8.8.1　鼠标器 …… 262
8.8.2　其他定位设备 …… 263
8.8.3　扫描仪 …… 264
8.9　打印输出设备 …… 265
8.9.1　打印机概述 …… 266
8.9.2　打印机的主要性能指标 …… 266
8.9.3　针式打印机工作原理 …… 267
8.9.4　喷墨打印机工作原理 …… 269
8.9.5　激光打印机工作原理 …… 270
8.10　显示设备 …… 271
8.10.1　显示器概述 …… 271
8.10.2　CRT 显示器 …… 271
8.10.3　字符显示器的工作原理 …… 275
8.10.4　图形显示器的工作原理 …… 277
8.10.5　LCD 显示器 …… 279
8.10.6　视频显示标准 …… 281

8.10.7 微型计算机的显示适配器 …… 282
习题 …… 283

第9章 输入输出系统 …… 286

9.1 主机与外设的连接 …… 286
9.1.1 输入输出接口 …… 286
9.1.2 接口的功能和基本组成 …… 287
9.1.3 外设的识别与端口寻址 …… 289
9.1.4 输入输出信息传送控制方式 …… 290
9.2 程序查询方式及其接口 …… 291
9.2.1 程序查询方式 …… 292
9.2.2 程序查询方式接口 …… 293
9.3 中断系统和程序中断方式 …… 295
9.3.1 中断的基本概念 …… 295
9.3.2 中断请求和中断判优 …… 297
9.3.3 中断响应和中断处理 …… 300
9.3.4 多重中断与中断屏蔽 …… 303
9.3.5 中断全过程 …… 307
9.3.6 程序中断接口结构 …… 307
9.4 DMA方式及其接口 …… 308
9.4.1 DMA方式的基本概念 …… 308
9.4.2 DMA接口 …… 310
9.4.3 DMA传送方法与传送过程 …… 312
9.5 通道控制方式 …… 314
9.5.1 通道的基本概念 …… 314
9.5.2 通道的类型与结构 …… 315
9.5.3 通道程序 …… 317
9.5.4 通道工作过程 …… 320
习题 …… 320

索引 …… 323

参考文献 …… 333

第1章 概论

本章从存储程序的概念开始，讨论计算机的基本组成与工作原理，使读者对于计算机系统先有一个简单的整体概念，为今后深入学习各个部件打下基础。

1.1 电子计算机与存储程序控制

电子计算机是一种不需要人工直接干预，能够自动、高速、准确地对各种信息进行处理和存储的电子设备。电子计算机从总体上来说可以分为两大类：电子模拟计算机和电子数字计算机。电子模拟计算机中处理的信息是连续变化的物理量，运算的过程也是连续的；而电子数字计算机中处理的信息是在时间上离散的数字量，运算的过程是不连续的。通常所说的计算机都是指电子数字计算机。

1.1.1 电子计算机的发展

1. 计算机的发展历史

人们习惯把电子计算机的发展历史分“代”，其实分代并没有统一的标准。最常见的分代方法是根据计算机所采用的电子器件来划分的：

第一代，1946—1958年，电子管计算机；

第二代，1958—1964年，晶体管计算机；

第三代，1964—1971年，小、中规模集成电路(SSI、MSI)计算机；

第四代，1971—至今，大、超大规模集成电路(LSI、VLSI)计算机。

(1) 电子管计算机时代

这一时期的计算机采用电子管作为基本器件，初期使用延迟线作为存储器，以后发明了磁芯存储器。早期的计算机主要用于科学计算，为军事与国防尖端科技服务。

(2) 晶体管计算机时代

这一时期计算机的基本器件由电子管改为晶体管，存储器采用磁芯存储器。运算速度从每秒几千次提高到几十万次，存储器的容量从几千存储单元提高到10万存储单元以上。这不仅使计算机在军事与尖端技术上的应用范围进一步扩大，而且在气象、工程设计、数据处理以及其他科学研究领域也得到应用。

(3) 小、中规模集成电路计算机时代

这一时期的计算机采用小、中规模集成电路为基本器件,因此功耗、体积和价格等进一步下降,而速度及可靠性相应提高,使得计算机的应用范围进一步扩大。

(4) 大、超大规模集成电路计算机时代

20世纪60年代后,微电子技术发展迅猛,半导体存储器问世,迅速取代了磁芯存储器,并不断向大容量、高集成度、高速度方向发展。从1971年开始出现了包含CPU的单片集成电路(微处理器),以微处理器为核心的电子计算机就是微型计算机。微型计算机的出现,形成了计算机发展史上的又一次革命,使计算机进入了几乎所有的行业。

严格意义上说,现代计算机还属于第四代计算机,但是随着集成电路的不断发展,单片集成电路的规模越来越大。有专家将单片超过100万晶体管以上的集成电路称为特大规模集成电路(ULSI),单片达到一亿到十亿晶体管的集成电路称为极大规模集成电路(ELSI)。

2. 计算机的发展趋势

现在,世界已进入了计算机时代,计算机的发展趋势正向着“两极”分化。一极是微型计算机向更微型化、网络化、高性能、多用途方向发展。微型计算机分为台式机、便携机、笔记本机、亚笔记本机、掌上机等。由于它们体积小、成本低而占领了整个国民经济和社会生活的各个领域。另一极则是巨型机向更巨型化、超高速、并行处理、智能化方向发展。它是一个国家科技水平、经济实力、军事实力的象征。在解决天气预报、地震分析、航空气动、流体力学、卫星遥感、激光武器、海洋工程等方面的问题上,巨型机将大显身手。

随着新的元器件及其技术的发展,新型的超导计算机、量子计算机、光子计算机、生物计算机、纳米计算机等将会在不久的将来走进人们的生活,遍布各个领域。

1.1.2 存储程序概念

世界上第一台电子数字计算机是1946年2月在美国宾夕法尼亚大学诞生的ENIAC(Electronic Numerical Integrator and Computer)。ENIAC是一个庞然大物,它共用了18 000多个电子管,重达30t,占地面积170m^2,每秒钟可完成5000次加法运算。ENIAC有一个很大的缺点,即它的存储容量极小,只能存储20个字长为10位的十进制数,所以只能用线路连接的方法来编排程序,每次解题都要依靠人工来改变接线,准备时间将大大超过实际计算时间。

在研制ENIAC的同时,以美籍匈牙利裔数学家冯·诺依曼(John von Neumann)为首的研制小组提出了“存储程序控制”的计算机结构,并开始了存储程序控制的计算机EDVAC(Electronic Discrete Variable Automatic Computer)的研制。由于种种原因,EDVAC直到1951年才问世。而吸收了冯·诺依曼的设计思想,由英国剑桥大学研制的EDSAC(Electronic Delay Storage Automatic Computer)则先于它两年诞生,成为事实上的第一台存储程序的计算机。

存储程序概念是冯·诺依曼等人于1945年6月首先提出来的,它可以简要地概括为以下3点:

① 计算机(指硬件)应由运算器、存储器、控制器、输入设备和输出设备五大基本部件组成。

② 计算机内部采用二进制来表示指令和数据。

③ 将编好的程序和原始数据事先存入存储器中，然后再启动计算机工作，这就是存储程序的基本含义。

冯·诺依曼对计算机界的最大贡献在于“存储程序控制”概念的提出和实现。70多年来，虽然计算机的发展速度是惊人的，但就其结构原理来说，目前绝大多数计算机仍建立在存储程序概念的基础上。通常把符合“存储程序概念”的计算机统称为冯·诺依曼型计算机。当然，现代计算机与早期计算机相比在结构上还是有许多改进的。

随着计算机技术的不断发展，也暴露出了冯·诺依曼型计算机的主要弱点：存储器访问会成为瓶颈。目前已出现了一些突破存储程序控制的计算机，统称为非冯计算机，如数据驱动的数据流计算机、需求驱动的归约计算机和模式匹配驱动的智能计算机等。本书讨论的范围仅限于冯·诺依曼型计算机的组成，有关非冯计算机将在后续的课程中进行讨论。

1.2　计算机的硬件组成

原始的冯·诺依曼计算机在结构上是以运算器为中心的，而发展到现在，已转向以存储器为中心，图1-1为计算机最基本的组成框图。

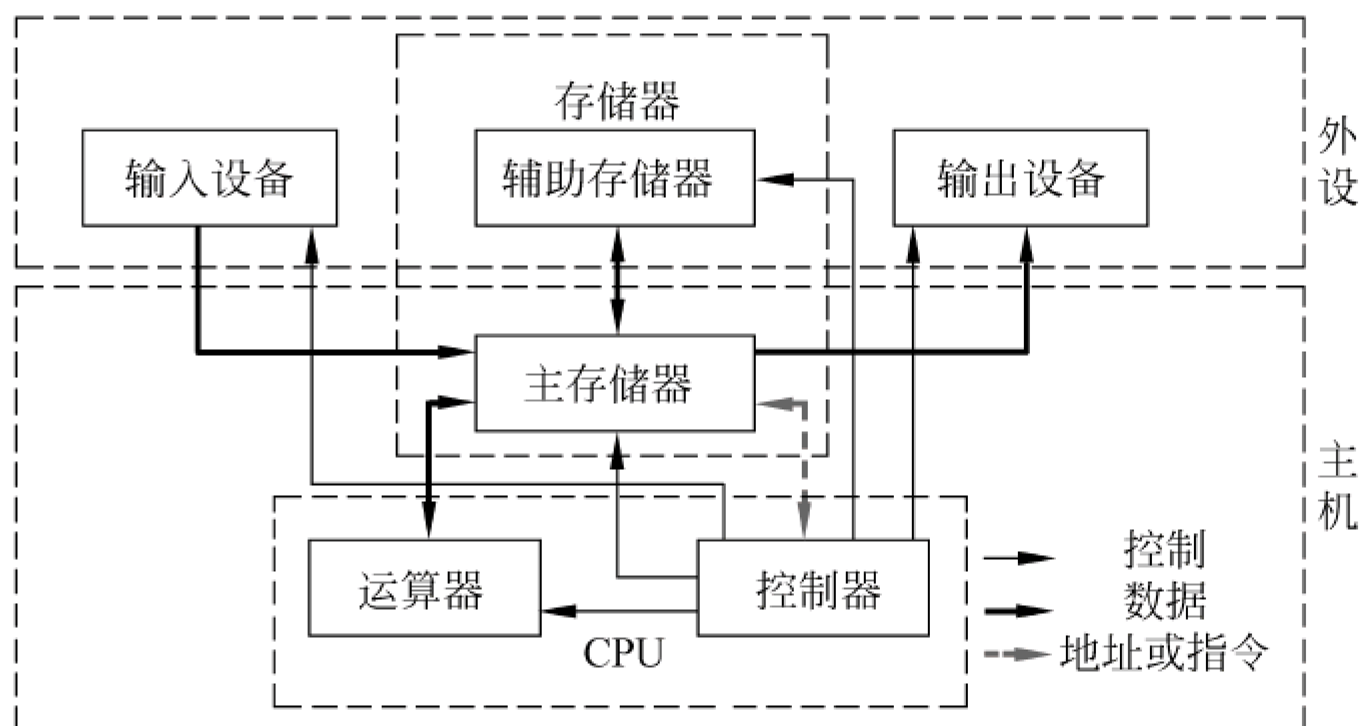


图1-1　计算机的组成

通常将运算器和控制器合称为中央处理器(Central Processing Unit，CPU)。在由超大规模集成电路构成的微型计算机中，往往将CPU制成一块芯片，称为微处理器。

中央处理器和主存储器(内存储器)一起组成主机部分。除去主机以外的硬件装置(如输入设备、输出设备和辅助存储器等)称为外围设备或外部设备。

1.2.1　计算机的主要部件

1. 输入设备

输入设备的任务是把人们编好的程序和原始数据送到计算机中，并且将它们转换成计算机内部所能识别和接受的信息方式。

按输入信息的形态输入可分为字符(包括汉字)输入、图形输入、图像输入及语音输入

等。目前,常见的输入设备有键盘、鼠标、扫描仪、摄像头、手写输入板等。辅助存储器(磁盘、磁带)也可以视为输入设备。

2. 输出设备

输出设备的任务是将计算机的处理结果以数字、字符(汉字)、图形、图像、声音等形式送出计算机。

常用的输出设备有打印机、显示器、绘图仪等。辅助存储器也可以视为输出设备。

3. 存储器

存储器是用来存放程序和数据的部件,它是一个记忆装置,是计算机能够实现"存储程序控制"的基础。

在计算机系统中,规模较大的存储器往往分成若干级,称为存储系统。图1-2所示的是常见的三级存储系统。主存储器可由CPU直接访问,存取速度快,但容量较小,一般用来存放当前正在执行的程序和数据。辅助存储器设置在主机外部,它的存储容量大,价格较低,但存取速度较慢,一般用来存放暂时不参与运行的程序和数据。CPU不可以直接访问辅存,辅存中的程序和数据在需要时才传送到主存,因此它是主存的补充和后援。当CPU速度很高时,为了使访问存储器的速度能与CPU的速度相匹配,又在主存和CPU间增设了一级Cache(高速缓冲存储器)。Cache的存取速度比主存更快,但容量更小,用来存放当前最急需处理的程序和数据,以便快速地向CPU提供指令和数据。

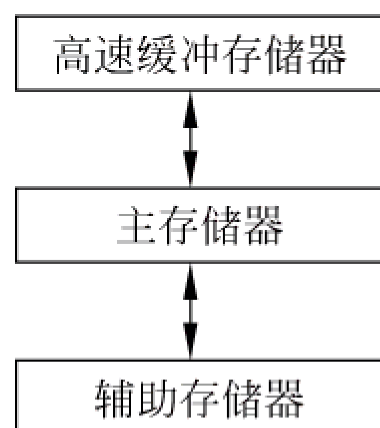


图1-2 三级存储系统

4. 运算器

运算器是对信息进行处理和运算的部件。经常进行的运算是算术运算和逻辑运算,所以运算器又称为算术逻辑运算部件(Arithmetic and Logical Unit,ALU)。

运算器的核心是加法器。运算器中还有若干个通用寄存器或累加寄存器,用来暂存操作数并存放运算结果。寄存器的存取速度比存储器的存取速度快得多。

5. 控制器

控制器是整个计算机的指挥中心,它的主要功能是按照人们预先确定的操作步骤,控制整个计算机的各部件有条不紊地自动工作。

控制器从主存中逐条地取出指令进行分析,根据指令的不同来安排操作顺序,向各部件发出相应的操作信号,控制它们执行指令所规定的任务。

控制器中包括一些专用的寄存器。

1.2.2 计算机各大部件之间连接

将前述的各大基本部件,按某种方式连接起来就构成了计算机的硬件系统。

1. 总线结构(小、微型机的典型结构)

所谓总线(Bus)是一组能为多个部件服务的公共信息传送线路,它能分时地发送与接收各部件的信息。计算机中采用总线结构,可以大大减少信息传送线的数量,又可以提高计算机扩充主存及外部设备的灵活性。

最简单的总线结构是单总线结构,如图1-3所示。各大部件都连接在单一的一组总线上,故将这个单总线称为系统总线。CPU与主存、CPU与外设之间可以直接进行信息交换,主存与外设、外设与外设之间也可以直接进行信息交换,而无须经过CPU的干预。

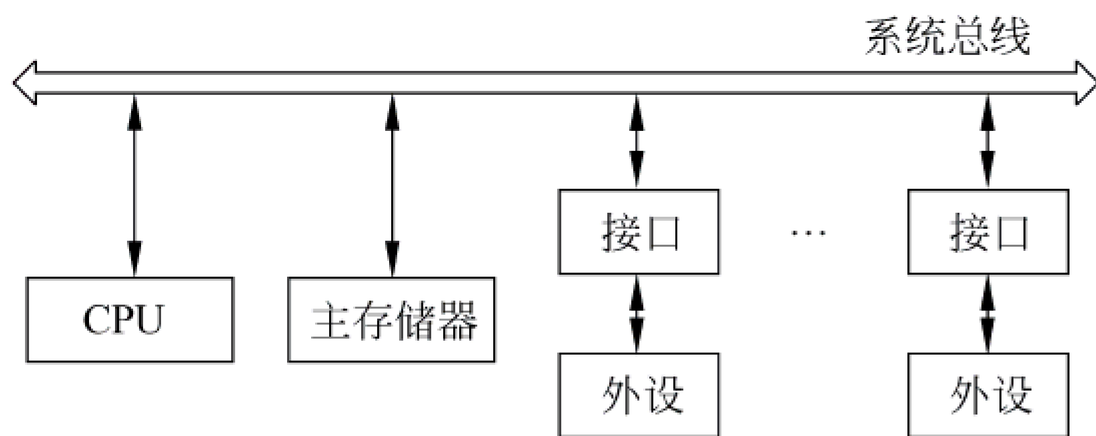


图1-3　单总线结构

单总线结构提高了CPU的工作效率,而且外设连接灵活,易于扩充。但由于所有部件都挂在同一组总线上,而总线又只能分时地工作,故同一时刻只允许一对设备(或部件)之间传送信息。

所谓单总线并不是指只有一根信号线。系统总线按传送信息的不同可以细分为地址总线、数据总线和控制总线。地址总线(Address Bus)由单方向的多根信号线组成,用于CPU向主存、外设传输地址信息;数据总线(Data Bus)由双方向的多根信号线组成,CPU可以沿这些线从主存或外设读入数据,也可以沿这些线向主存或外设送出数据;控制总线(Control Bus)上传输的是控制信息,包括CPU送出的控制命令和主存(或外设)返回CPU的反馈信号。

总线结构是小、微型计算机的典型结构。这是因为小、微型计算机的设计目标是以较小的硬件代价组成具有较强功能的系统,而总线结构正好能满足这一要求。

2. 大、中型计算机的典型结构

大、中型计算机系统的设计目标更着重于系统功能的扩大与效率的提高。图1-4为大、中型计算机的典型结构图。在系统连接上分为四级:主机、通道、设备控制器和外部设备。

通道是承担I/O操作管理的主要部件,每个通道可以接一台或几台设备控制器,每个设备控制器又可接一台或几台外部设备,这样整个系统就可以连接很多的外部设备。这种结构具有较大的扩充变化余地。对较小的系统来说,可将设备控制器与外设合并在一起,将通道与CPU合并在一起;对较大的系统,则单独设置通道部件;对更大的系统,通道可发展成为具有处理功能的外围处理机。

1.2.3　不同对象观察到的计算机硬件系统

从第一台计算机诞生至今,已经经历了70多年的风风雨雨,目前计算机可以说已经无

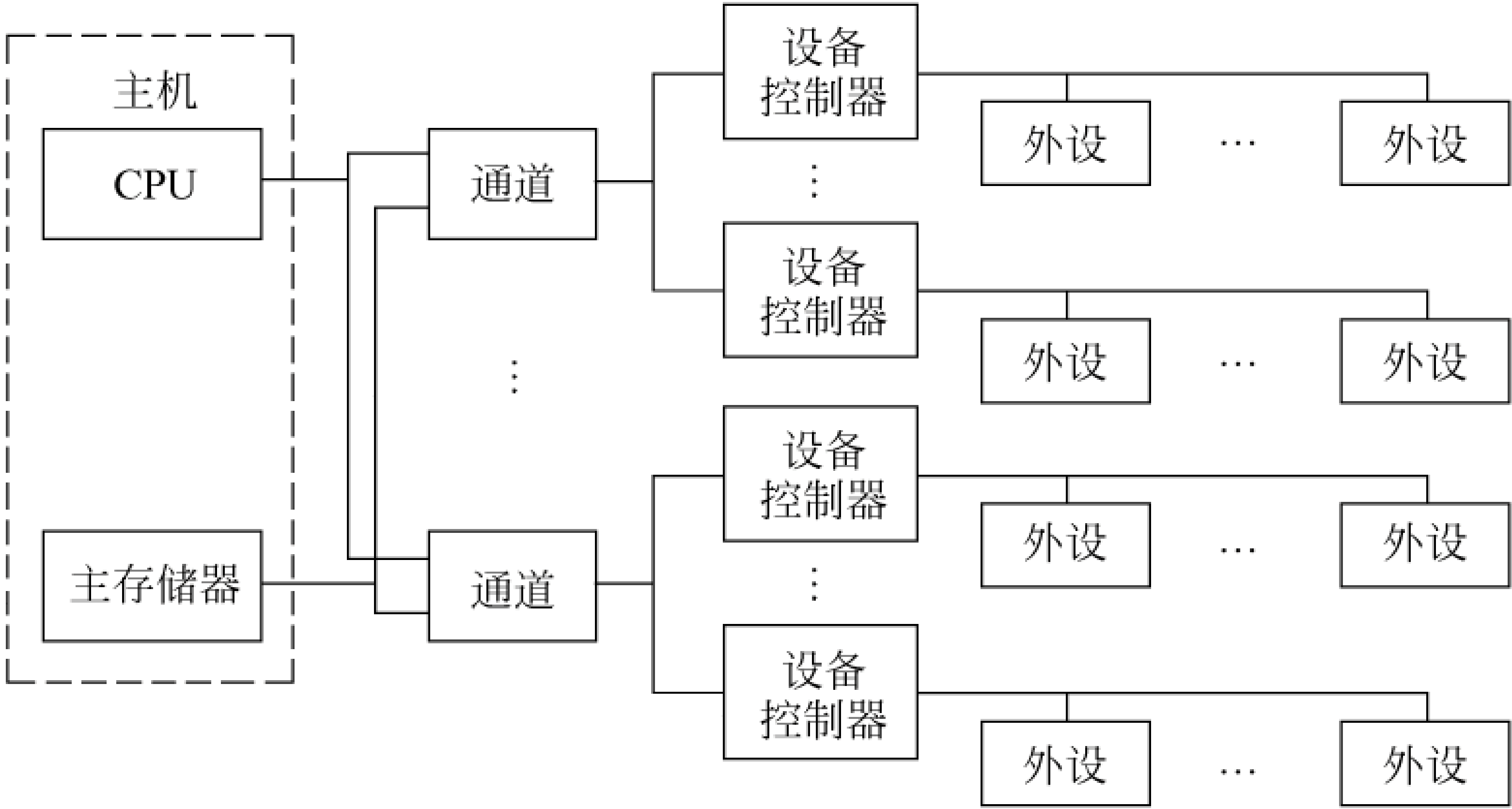


图 1-4　大、中型计算机的典型结构

处不在、无事不通。在绝大多数的人看来,计算机(主要指微型计算机)已不再是陌生和充满玄机的神秘之物,而仅仅是一种常用工具而已。然而不同对象所观察到的计算机硬件系统是不相同的,所涉及的技术问题也是不同层次的,下面仅以微型计算机为例来分析它们之间的区别。

1. 一般用户观察到的计算机硬件系统

一般用户是指那些仅局限于使用计算机最基本功能的用户。他们观察到的只是计算机的用户界面,如人机交互使用的键盘、鼠标、显示器,用于存储信息的磁盘、光盘等,计算机本身对于他们来说只是一个或立或卧在那里的铁箱子,至于它内部的结构和组成、工作原理等都是不必关心的。一般用户观察到的计算机硬件系统如图 1-5 所示。



图 1-5　一般用户观察到的计算机硬件系统

2. 专业用户观察到的计算机硬件系统

专业用户观察到的计算机硬件系统要比一般用户深入得多,他们可能更多地关注计算机机箱内各部分的结构和组成,专业用户观察到的计算机硬件系统如图 1-6 所示。

图 1-6 中除了键盘、显示器以外,其余部分都在机箱内。机箱中的核心是主板,微处理器、内存条、外部高速缓冲存储器(Cache)、显卡、磁盘控制器等可以直接制作在主板上或插在主板上。近年的计算机已都将外部 Cache 集成在微处理器芯片中了。

3. 计算机设计者观察到的计算机硬件系统

计算机的设计者更多地关心计算机的组成原理和实现方法,图 1-7 所示的是计算机设计者观察到的计算机硬件系统,此时计算机的硬件系统被进一步分解。

从图 1-7 可以看出,计算机硬件系统是由运算器、控制器、存储器和外设组成的,这就是系统级。系统级是由寄存器级组成的,寄存器级又是由门级组成的,门级最后由晶体

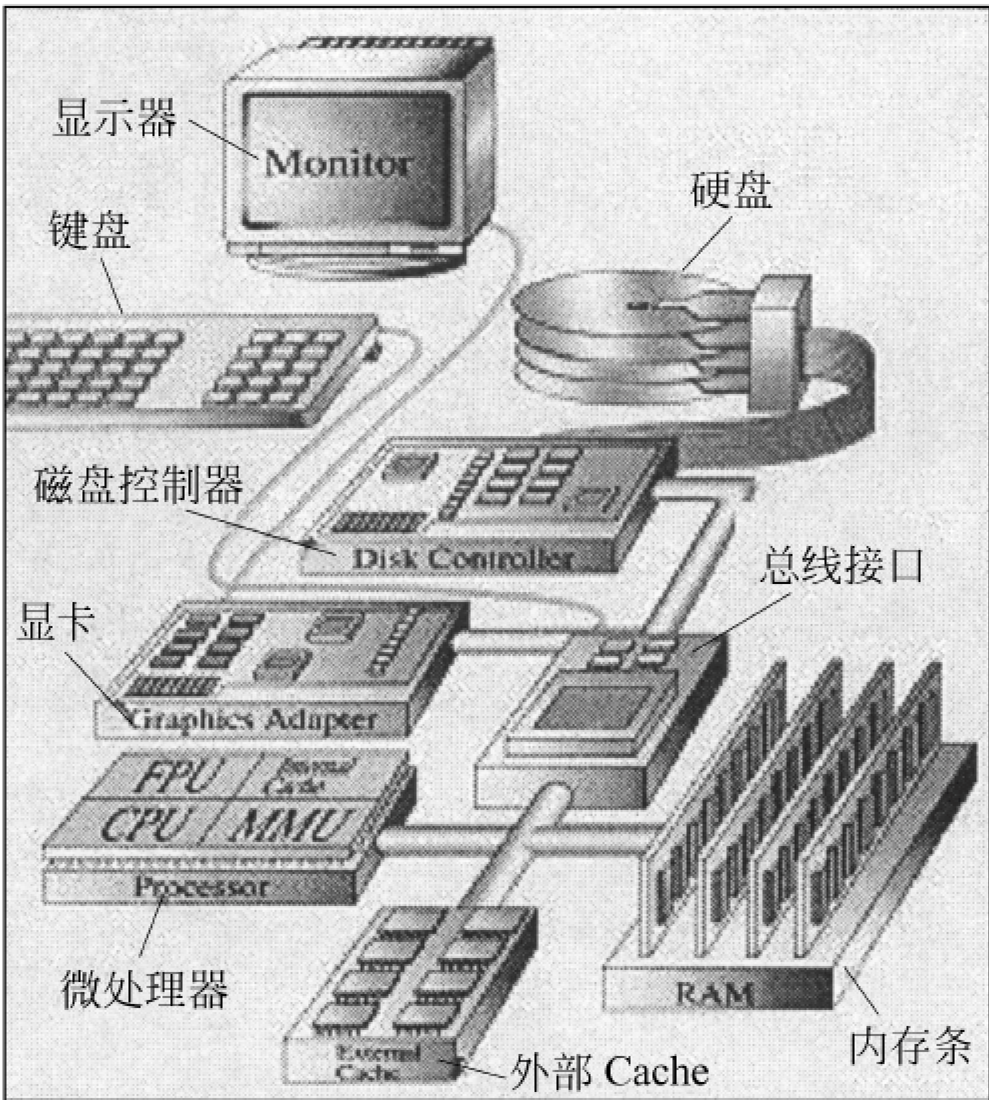


图 1-6　专业用户观察到的计算机硬件系统

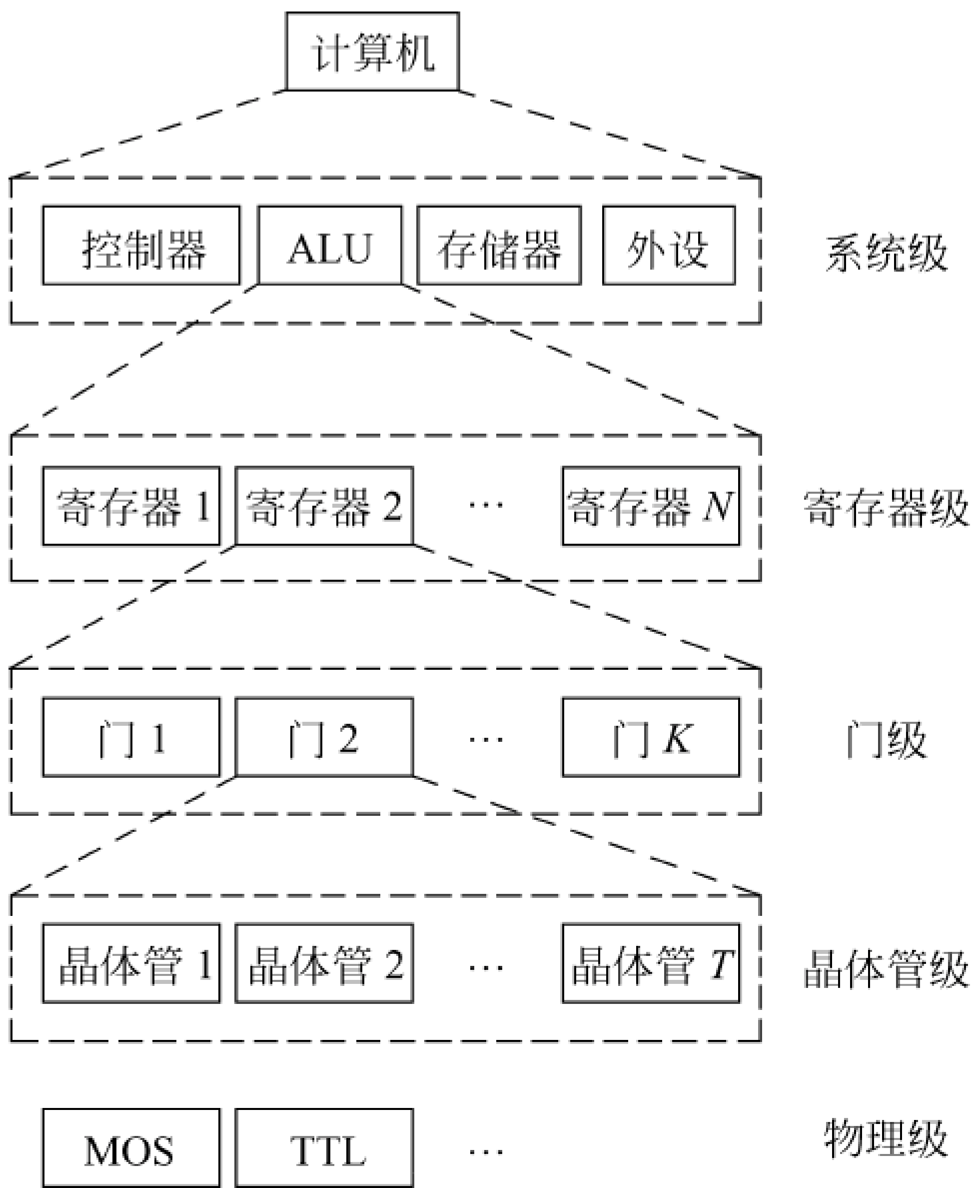


图 1-7　计算机设计者观察到的计算机硬件系统

管级组成，而晶体管的物理实现主要有两种：金属氧化物半导体（MOS）和晶体管-晶体管逻辑（TTL）。

1.2.4　冯·诺依曼结构和哈佛结构的存储器设计思想

前面已经提到过冯·诺依曼计算机的存储程序概念，然而根据程序（指令序列）和数据的存放形式，存储器设计思想又可以分为冯·诺依曼结构和哈佛结构。

1. 冯·诺依曼结构

冯·诺依曼结构也称普林斯顿结构,是一种传统的存储器设计思想,即指令和数据是不加区别地混合存储在同一个存储器中的,共享数据总线,如图1-8所示。指令地址和数据地址指向同一个存储器的不同物理位置,指令和数据的宽度相同。由于指令和数据存放在同一存储器中,因此冯·诺依曼结构中不能同时取指令和取操作数。又由于存储器存取速度远远低于CPU运算速度,从而使计算机运算速度受到很大限制,CPU与共享存储器间的信息交换成了影响高速计算和系统性能的"瓶颈"。

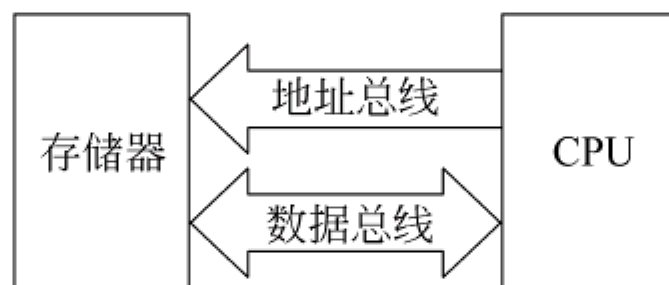


图1-8 冯·诺依曼结构的存储器设计

使用冯·诺依曼结构的中央处理器是很多的。例如,Intel公司的80x86及其他中央处理器、ARM公司的ARM7和MIPS公司的MIPS等也都采用了冯·诺依曼结构。

2. 哈佛结构

冯·诺依曼结构在面对高速、实时处理时,不可避免会造成总线拥挤。为此,哈佛大学提出了与冯·诺依曼结构完全不同的另一种存储器设计思想,人们习惯称之为哈佛结构。哈佛结构的指令和数据是完全分开的,存储器分为两部分,一个是程序存储器,用来存放指令,另一个是数据存储器,用来存放数据。哈佛结构至少有两组总线:程序存储器(PM)的数据总线和地址总线,数据存储器(DM)的数据总线和地址总线,如图1-9所示。这种分离的程序总线和数据总线,可允许同时获取指令字(来自程序存储器)和操作数(来自数据存储器)而互不干扰。这意味着在一个机器周期内可以同时准备好指令和操作数,本条指令执行时可以预先读取下一条指令,所以哈佛结构的中央处理器通常具有较高的执行效率。同时,由于指令和数据分开存放,可以使指令和数据有不同的数据宽度,如Microchip公司的PIC16芯片的指令宽度为14位,而数据宽度为8位。

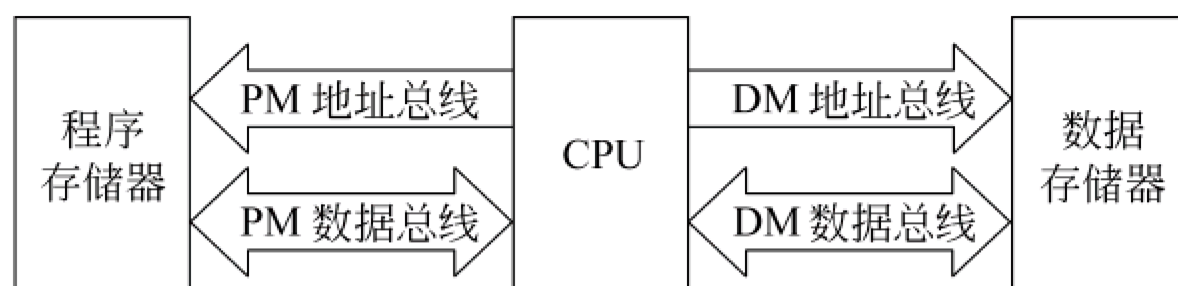


图1-9 哈佛结构的存储器设计

使用哈佛结构的中央处理器有很多。例如,Motorola公司的MC68系列,Zilog公司的Z8系列,以及ARM公司的ARM9、ARM10和ARM11。大多数单片机和数字信号处理(DSP)系统都使用哈佛结构。

目前,许多现代微型计算机中的高速缓冲存储器(Cache)采用哈佛结构,将Cache分为指令Cache和数据Cache两个部分,而主存储器采用冯·诺依曼结构,只有一个,由指令和数据合用。如此将冯·诺依曼结构和哈佛结构结合起来,不仅可以提高主存储器的利用率,而且可以提高程序执行的效率,缩短指令执行的时钟周期。

1.3 计算机系统

一个完整的计算机系统包含硬件系统和软件系统两大部分。硬件通常是指一切看得见、摸得到的设备实体；软件通常是泛指各类程序和文件，它们实际上是由一些算法以及其在计算机中的表示所构成的。

1.3.1 硬件与软件的关系

硬件是计算机系统的物质基础，正是在硬件高度发展的基础上，才有软件赖以生存的空间和活动场所，没有硬件对软件的支持，软件的功能就无从谈起；同样，软件是计算机系统的灵魂，没有软件的硬件“裸机”将不能提供给用户使用，犹如一堆废铁。因此，硬件和软件是相辅相成、不可分割的整体。

当前，计算机的硬件和软件正朝着互相渗透、互相融合的方向发展，在计算机系统中没有一条明确的硬件与软件的分界线。原来一些由硬件实现的功能可以改由软件模拟来实现，这种做法称为硬件软化，它可以增强系统的功能和适应性；同样，原来由软件实现的功能也可以改由硬件来实现，称为软件硬化，它可以显著降低软件在时间上的开销。由此可见，硬件和软件之间的界面是浮动的，对于程序设计人员来说，硬件和软件在逻辑上是等价的。一项功能究竟采用何种方式实现，应该从系统的效率、速度、价格和资源状况等诸多方面综合考虑。

既然硬件和软件不存在一条固定的一成不变的界线，那么今天的软件可能就是明天的硬件，今天的硬件也可能就是明天的软件。

除去硬件和软件，还有一个概念需要引起注意，这就是固件(Firmware)。固件一词是1967年由美国人A·Opler首先提出来的。固件是指那些存储在能永久保存信息的器件(如ROM)中的程序，是具有软件功能的硬件。固件的性能指标介于硬件与软件之间，吸收了软、硬件各自的优点，其执行速度快于软件，灵活性优于硬件，是软、硬件结合的产物，计算机功能的固件化将成为计算机发展的一个趋势。

1.3.2 系列机和软件兼容

计算机技术是飞速发展的技术，随着元器件、硬件技术和工业生产能力的迅猛发展，新的高性能的计算机不断地被研制和生产出来。用户希望在新的计算机系统出台后，原先已开发的软件仍能继续在升级换代后的新型号的机器上使用，这就要求软件具有可兼容性。

所谓系列机，是指一个厂家生产的，具有相同的系统结构，但具有不同组成和实现的一系列不同型号的机器。

系列机从程序设计者的角度看具有相同的机器属性，即相同的系统结构。这里的相同是指在指令系统、数据格式、字符编码、中断系统、控制方式和输入输出操作方式等多个方面保持统一，从而保证了软件的兼容。系列机的软件兼容分为向上兼容、向下兼容、向前兼容和向后兼容4种。向上(下)兼容指的是按某档次机器编制的程序，不加修改就能运行在比它更高(低)档的机器上；向前(后)兼容是指按某个时期投入市场的某种型号机器编制的程

序,不加修改就能运行在它之前(后)投入市场的机器上。图1-10形象地说明了兼容性的概念。对系列机的软件向下和向前兼容可以不作要求,但必须保证向后兼容,力争做到向上兼容。

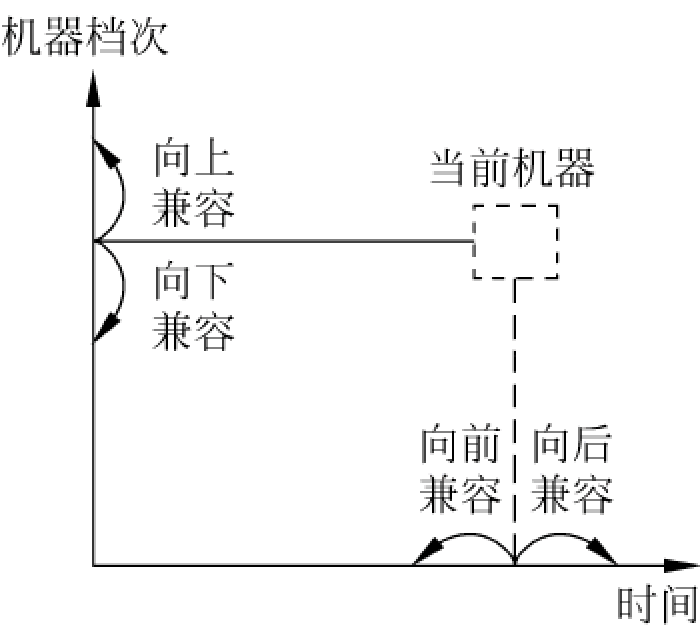


图1-10 兼容性示意

1.3.3 计算机系统的多层次结构

现代计算机系统是一个硬件与软件组成的综合体,可以把它看作按功能划分的多级层次结构,如图1-11所示。

第零级是硬联逻辑级,这是计算机的内核,由门、触发器等逻辑电路组成。

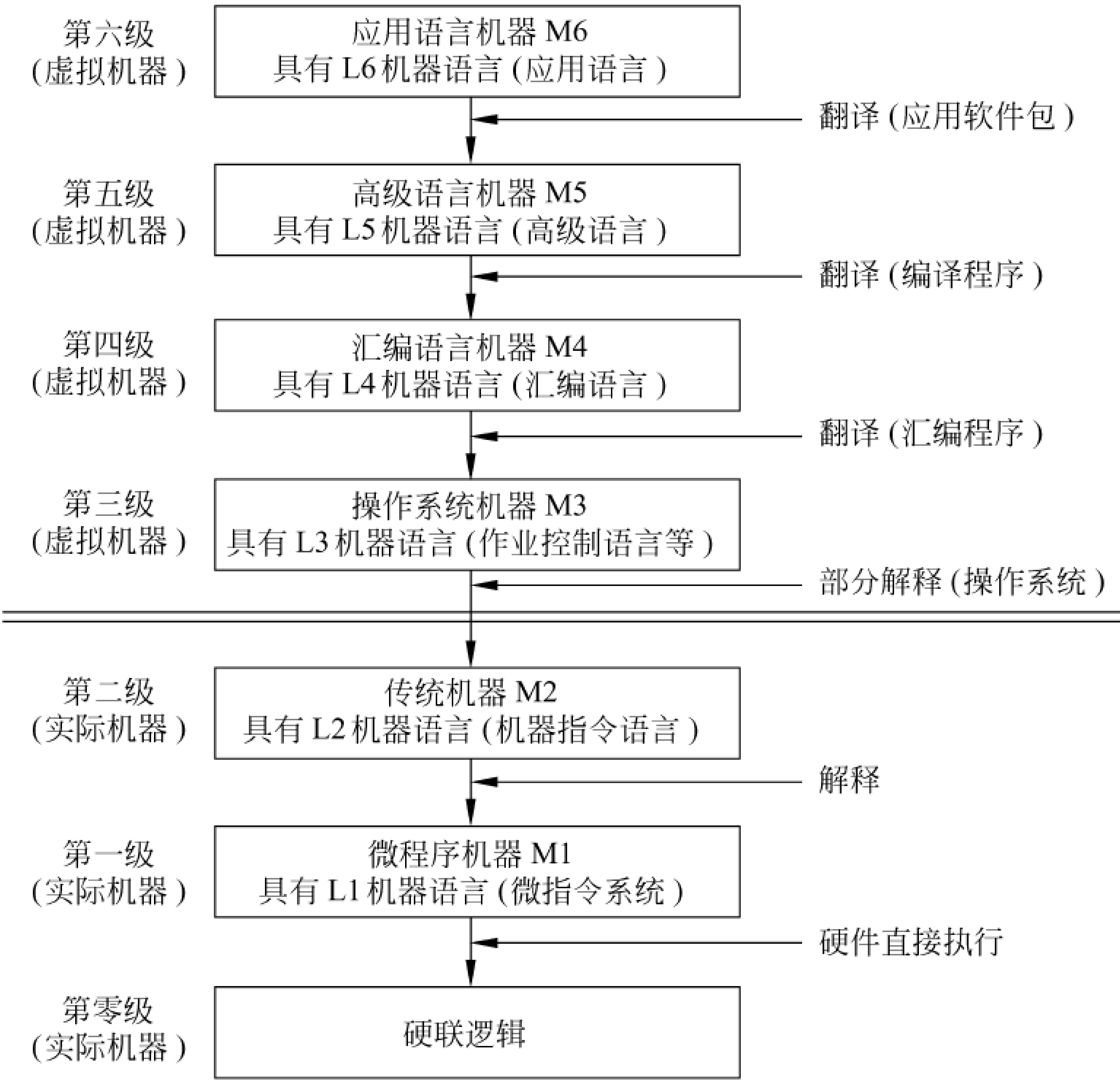


图1-11 计算机系统的多层次结构

第一级是微程序机器级。这级的机器语言是微指令集,用微指令编写的微程序一般是直接由硬件执行的。

第二级是传统机器级。这级的机器语言是该机的指令集,用机器指令编写的程序可以由微程序进行解释。

第三级是操作系统机器级。从操作系统的基本功能来看,一方面它要直接管理传统机器中的软硬件资源,另一方面它又是传统机器的延伸。

第四级是汇编语言机器级。这级的机器语言是汇编语言,完成汇编语言翻译的程序称为汇编程序。

第五级是高级语言机器级。这级的机器语言就是各种高级语言，通常用编译程序来完成高级语言翻译的工作。

第六级是应用语言机器级。这一级是为了使计算机满足某种用途而专门设计的，因此这一级语言就是各种面向问题的应用语言。

把计算机系统按功能划分成多级层次结构，有利于正确理解计算机系统的工作过程，明确软件、硬件在计算机系统中的地位和作用。

1.3.4　实际机器和虚拟机器

在图 1-11 所示的多级层次结构中，对每一个机器级的用户来说，都可以将此机器级看作一台独立的使用自己特有的“机器语言”的机器。

实际机器是指由硬件或固件实现的机器，如图 1-11 所示的第零级到第二级。虚拟机器是指以软件或以软件为主实现的机器，如图 1-11 所示的第三级到第六级。

虚拟机器只对该级的观察者存在，即在某一级观察者看来，他只需要通过该级的语言来了解和使用计算机，至于下级是如何工作和实现就不必关心了。如高级语言级机器及应用语言机器级的用户，不必了解机器的具体组成，不必熟悉指令系统，直接用所指定的语言描述所要解决的问题即可。

1.4　计算机的工作过程和主要性能指标

为使计算机按预定要求工作，首先要编制程序。程序是一个特定的指令序列，它告诉计算机要做哪些事，按什么步骤去做。指令是一组二进制信息的代码，用来表示计算机所能完成的基本操作。

1.4.1　计算机的工作过程

编制好的程序放在主存中，由控制器控制逐条取出指令执行，下面以一个例子来加以说明。

例如：计算 $a+b-c=?$（设 a、b、c 为已知的 3 个数，分别存放在主存的 5～7 号单元中，结果将存放在主存的 8 号单元），如果采用单累加寄存器结构的运算器，完成上述计算至少需要 5 条指令，这 5 条指令依次存放在主存的 0～4 号单元中，参加运算的数也必须存放在主存指定的单元中，主存中有关单元的内容如图 1-12(a)所示。运算器的简单框图如图 1-12(b)所示，参加运算的两个操作数一个来自累加寄存器，一个来自主存，运算结果则放在累加寄存器中。图 1-12(b)中的存储器数据寄存器是用来暂存从主存中读出的数据或写入主存的数据的，它本身不属于运算器的范畴。

计算机的控制器将控制指令逐条地执行，最终得到正确的结果，步骤如下：

① 执行取数指令，从主存 5 号单元取出数 a，送入累加寄存器中。

② 执行加法指令，将累加寄存器中的内容 a 与从主存 6 号单元取出的数 b 一起送到 ALU 中相加，结果 $a+b$ 保留在累加寄存器中。

③ 执行减法指令，将累加寄存器中的内容 $a+b$ 与从主存 7 号单元取出的数 c 一起送到 ALU 中相减，结果 $a+b-c$ 保留在累加寄存器中。

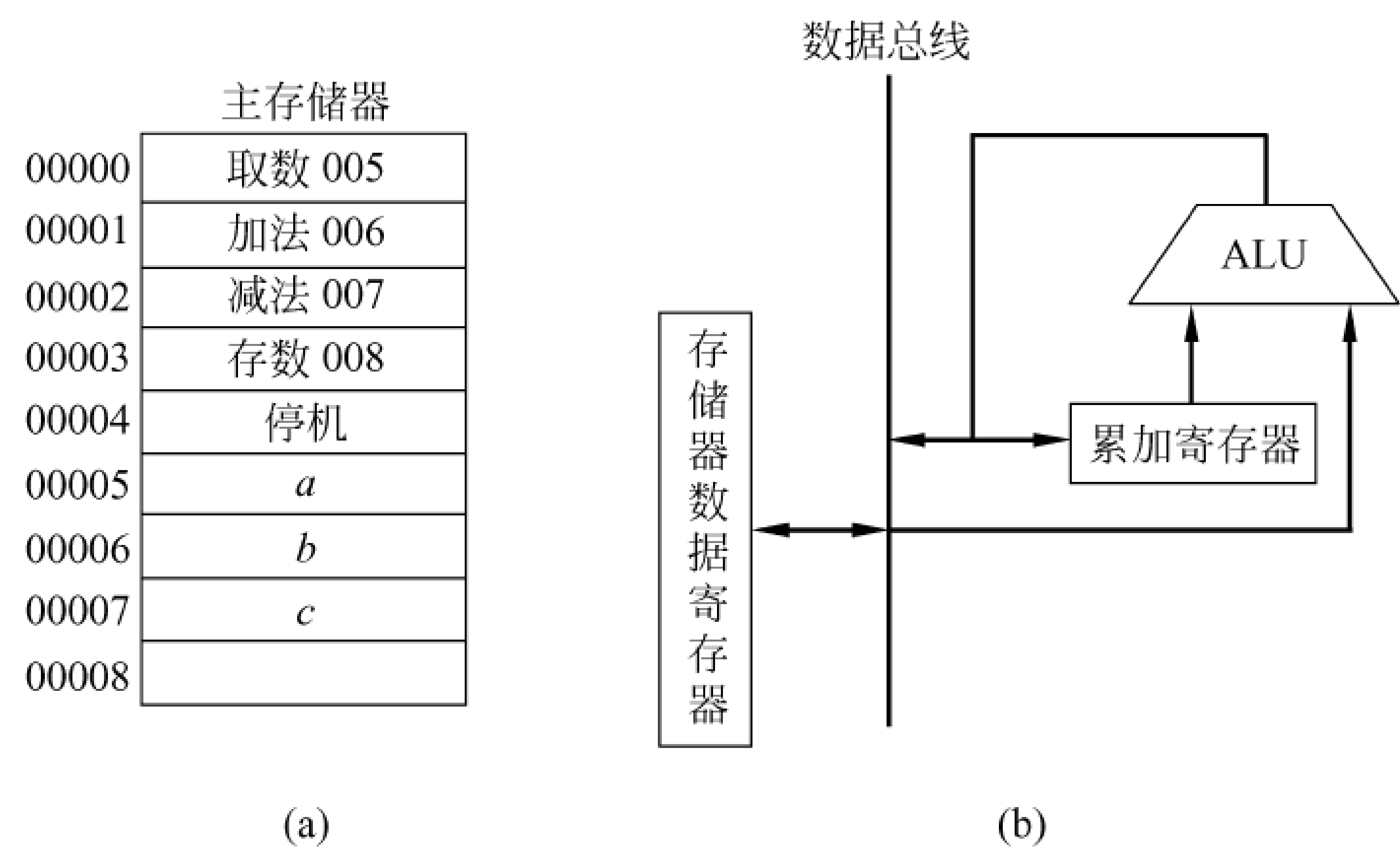


图 1-12　计算机执行过程实例

④ 执行存数指令,把累加寄存器的内容 $a+b-c$ 存至主存 8 号单元。

⑤ 执行停机指令,计算机停止工作。

1.4.2　计算机的主要性能指标

为了进一步了解计算机的特性,全面衡量一台计算机的性能,下面介绍计算机的主要性能指标。

1. 机器字长

机器字长是指参与运算的数的基本位数,它是由加法器、寄存器的位数决定的,所以机器字长一般等于内部寄存器的大小。字长标志着精度,字长越长,计算的精度就越高。

在计算机中为了更灵活地表达和处理信息,以字节(Byte)为基本单位,字节用大写字母B表示。一个字节等于8位二进制位(bit),位用小写字母b表示。通常所说的字(Word)是指数据字,不同的计算机,数据字长度可以不相同,但对于系列机来说,数据字的长度应该是固定的。例如,Intel 80x86系列中,一个数据字等于16位;IBM 303X系列中,一个数据字等于32位。

需要注意的是,这里所说的字(数据字)和字长(机器字长)的概念是有区别的,字实际上是一个度量单位,用来度量各种数据类型的宽度,而字长表示数据运算的宽度,反映了计算机处理信息的能力。它们两者的长度可以一样,也可以不一样。

2. 数据通路宽度

数据总线一次所能并行传送信息的位数,称为数据通路宽度。它影响到信息的传送能力,从而影响计算机的有效处理速度。这里所说的数据通路宽度是指外部数据总线的宽度,它与CPU内部的数据总线宽度(内部寄存器的位数)有可能不同。有些CPU的内、外数据总线宽度相等,如Intel 8086、80286、80486等;有些CPU的外部数据总线宽度小于内部,如8088、80386SX等;也有些CPU的外部数据总线宽度大于内部,如Pentium等。所有的Pentium都有64位外部数据总线和32位内部寄存器——这一结构看起来似乎有问题,这

是因为 Pentium 有两条 32 位流水线，它就像两个合在一起的 32 位芯片，64 位数据总线可以满足高效地充满多个寄存器的需要。

3. 主存容量

一个主存储器所能存储的全部信息量称为主存容量。通常，以字节数来表示存储容量，这样的计算机称为字节编址的计算机。也有一些计算机是以字为单位编址的，它们用字数乘以字长来表示存储容量。在表示容量大小时，经常用到 K、M、G、T、P 之类的字符，它们与通常意义上的 K、M、G、T、P 有些差异，如表 1-1 所示。

表 1-1　K、M、G、T、P 的定义

单　位	通常意义	实际表示
K(Kilo)	10^3	$2^{10}=1024$
M(Mega)	10^6	$2^{20}=1\ 048\ 576$
G(Giga)	10^9	$2^{30}=1\ 073\ 741\ 824$
T(Tera)	10^{12}	$2^{40}=1\ 099\ 511\ 627\ 776$
P(Peta)	10^{15}	$2^{50}=1\ 125\ 899\ 906\ 842\ 624$

1024 个字节称为 1KB，1024KB 称为 1MB，1024MB 称为 1GB……计算机的主存容量越大，存放的信息就越多，处理问题的能力就越强。

4. 运算速度

计算机的运算速度与许多因素有关，如机器的主频、执行什么样的操作以及主存本身的速度等。

(1) 吞吐量和响应时间

吞吐量是指系统在单位时间内处理请求的数量。响应时间是指系统对请求作出响应的时间，响应时间包括 CPU 时间（运行一个程序所花费的时间）与等待时间（用于磁盘访问、存储器访问、I/O 操作、操作系统开销等时间）的总和。

(2) 主频和 CPU 时钟周期

CPU 的主频又称为时钟频率，表示在 CPU 内数字脉冲信号振荡的速度。主频的倒数就是 CPU 时钟周期，这是 CPU 中最小的时间元素。每个动作至少需要一个时钟周期。

(3) CPI

CPI(Cycles per Instruction)是指每条指令执行所用的时钟周期数。在现代高性能计算机中，由于采用各种并行技术，使指令执行高度并行化，常常是一个系统时钟周期内可以处理若干条指令，所以 CPI 参数经常用 IPC(Instructions per Cycle)表示，即每个时钟周期执行的指令数。

$$\text{IPC}=\frac{1}{\text{CPI}}$$

(4) CPU 执行时间

$$\text{CPU 执行时间}=\frac{\text{CPU 时钟周期数}}{\text{时钟频率}}=\frac{\text{指令数}\times\text{CPI}}{\text{时钟频率}}$$

(5) MIPS 和 MFLOPS

MIPS(Million Instructions per Second)表示每秒执行多少百万条指令。对于一个给定的程序,MIPS 定义为:

$$\text{MIPS}=\frac{\text{指令条数}}{\text{执行时间}\times 10^{6}}=\frac{\text{主频}}{\text{CPI}}=\text{主频}\times\text{IPC}$$

这里所说的指令一般是指加、减运算这类短指令。

MFLOPS(Million Floating-point Operations per Second)表示每秒执行多少百万次浮点运算。对于一个给定的程序,MFLOPS 定义为:

$$\text{MFLOPS}=\frac{\text{浮点操作次数}}{\text{执行时间}\times 10^{6}}$$

MFLOPS 适用于衡量向量机的性能。

习　题

1-1　电子数字计算机和电子模拟计算机的区别在哪里?

1-2　冯·诺依曼计算机的特点是什么?其中最主要的一点是什么?

1-3　计算机的硬件是由哪些部件组成的?它们各有哪些功能?

1-4　什么叫总线?简述单总线结构的特点。

1-5　简述计算机的层次结构,说明各层次的主要特点。

1-6　计算机系统的主要技术指标有哪些?

第 2 章 数据的机器层次表示

数据是计算机加工和处理的对象，数据的机器层次表示将直接影响到计算机的结构和性能。本章主要介绍无符号数和带符号数的表示方法、数的定点与浮点表示方法、字符和汉字的编码方法、数据校验码等。熟悉和掌握本章的内容是学习计算机原理的最基本要求。

2.1 数值数据的表示

在计算机中，采用数字化方式来表示数据，数据有无符号数和带符号数之分，其中带符号数根据其编码的不同又有原码、补码和反码 3 种表示形式。

2.1.1 计算机中的数值数据

人们在日常生活中最常使用的是十进制数，这恐怕和人有十个手指头是分不开的。然而，在计算机中数据通常用二进制数来表示，任何数值数据都可以由一串“0”或“1”的数字构成。考虑到二进制数位数比较长，书写起来不方便，在计算机中也使用八进制和十六进制来表示数值数据。

为了避免出现误会，在给出一个数的同时就必须指明这个数的数制。例如，$(1010)_2$、$(1010)_8$、$(1010)_{10}$、$(1010)_{16}$所代表的数值就不同。除了用下标来表示不同的数制以外，在计算机中还常用后缀字母来表示不同的数制。后缀 B 表示这个数是二进制数(Binary)；后缀 Q 表示这个数是八进制数(Octal)，本来八进制数的英文单词的第一个字母应当是 O，因为字符 O 与数字 0 很容易混淆，所以常使用字符 Q 作为八进制数的后缀；后缀 H 表示这个数是十六进制数(Hexadecimal)；而后缀 D 表示这个数是十进制数(Decimal)。十进制数在书写时后缀 D 可以省略，其他进制在书写时后缀一般不可省略。例如，有 4 个数分别为 375D、101B、76Q、A17H，从后缀字母就可以知道它们分别是十进制数、二进制数、八进制数和十六进制数。

计算机系统设计师和程序员更钟情于采用程序设计语言的记号来表示不同进制的数，这就是前缀表示法。例如，在 C 语言中，八进制常数以前缀 0 开始，十六进制常数以前缀 0x 开始。

2.1.2 无符号数和带符号数

所谓无符号数，就是整个机器字长的全部二进制位均表示数值位(没有符号位)，相当于

数的绝对值。例如:

N_1=01001 表示无符号数 9。

N_2=11001 表示无符号数 25。

机器字长为 $n+1$ 位的无符号数的表示范围是 $0\sim(2^{n+1}-1)$,此时二进制的最高位也是数值位,其权值等于 2^n。若字长为 8 位,则数的表示范围为 0~255。

一般计算机中都设置有一些无符号数的运算和处理指令。例如,Intel 8086 中的 MUL 和 DIV 指令就是无符号数的乘法和除法指令,还有一些条件转移指令也是专门针对无符号数的。

然而,大量用到的数据还是带符号数,即正、负数。在日常生活中用符号+、−加绝对值来表示数值的大小,用这种形式表示的数值在计算机技术中称为“真值”。

对于数的符号+或−,计算机是无法识别的,因此需要把符号数码化。通常,约定二进制数的最高位为符号位,0 表示正号,1 表示负号。这种在计算机中使用的表示数的形式称为机器数,常见的机器数有原码、反码、补码等不同的表示形式。

带符号数的最高位被用来表示符号位,而不再表示数值位。前例中的 N_1、N_2 在这里的含义变为:

N_1=01001 表示+9。

N_2=11001 根据机器数的不同形式表示不同的值,如是原码则表示−9,若是补码则表示−7,若是反码则表示−6。

为了能正确地区别出真值和各种机器数,本书用 X 表示真值,$[X]_{原}$ 表示原码,$[X]_{补}$ 表示补码,$[X]_{反}$ 表示反码。

2.1.3 原码表示法

原码表示法是一种最简单的机器数表示法,其最高位为符号位,符号位为 0 时表示该数为正,符号位为 1 时表示该数为负,数值部分与真值相同。

若真值为纯小数,它的原码形式为 $X_s.X_1X_2\cdots X_n$,其中 X_s 表示符号位。原码的定义为:

$$[X]_{原}=\begin{cases}X & 0\leqslant X<1\\ 1-X=1+|X| & -1<X\leqslant 0\end{cases}$$

假设机器数长度为 5 位,则有:

例 2-1 $X=0.0110$ $[X]_{原}=X=0.0110$

$X=-0.0110$ $[X]_{原}=1-X=1-(-0.0110)=1+0.0110=1.0110$

若真值为纯整数,它的原码形式为 $X_sX_1X_2\cdots X_n$,其中 X_s 表示符号位。原码的定义为:

$$[X]_{原}=\begin{cases}X & 0\leqslant X<2^n\\ 2^n-X=2^n+|X| & -2^n<X\leqslant 0\end{cases}$$

例 2-2 $X=1101$ $[X]_{原}=X=01101$

$X=-1101$ $[X]_{原}=2^n-X=2^4-(-1101)=10000+1101=11101$

在原码表示中,真值 0 有两种不同的表示形式:

$$[+0]_{原}=00000$$

$$[-0]_{原}=10000$$

原码表示法的优点是直观易懂，机器数和真值间的相互转换很容易，用原码实现乘、除运算的规则很简单；缺点是实现加、减运算的规则较复杂。

2.1.4 补码表示法

1. 模和同余

为了理解补码表示法，首先需要引入模和同余的概念。

模(Module)是指一个计量器的容量，可用 M 表示。例如，一个4位的二进制计数器，当计数器从0计到15之后，再加1，计数值又变为0。这个计数器的容量 $M=2^4=16$，即模为16。由此可见，纯小数的模为2，一个字长为 $n+1$ 位的纯整数的模为 2^{n+1}。

同余概念是指两个整数 A 和 B 除以同一个正整数 M，所得余数相同，则称 A 和 B 对 M 同余，即 A 和 B 在以 M 为模时是相等的，可写成如下形式：

$$A=B\ (\bmod M)$$

对钟表而言，其模 $M=12$，故4点和16点、5点和17点……均是同余的，它们可以写成如下形式：

$$4=16\ (\bmod 12),\quad 5=17\ (\bmod 12)$$

利用模和同余概念的补码表示法在进行算术运算时可以使减法运算转化成加法运算，从而简化机器的运算器电路。

假设时钟停在8点，而现在正确的时间是6点，这时拨准时钟的方法有如下两种：

① 将分针倒着旋转两圈(即时钟倒拨2小时)$8-2=6$(做减法)。

② 将分针正着旋转10圈(即时钟正拨10小时)$8+10=6\ (\bmod 12)$(做加法)。

此时，
$$8-2=8+10\ (\bmod 12)$$

设：
$$A=-2,\quad B=10$$

则：
$$\frac{10}{12}=\frac{12-2}{12}=1+\frac{-2}{12}$$

故-2和10同余。同余的两个数具有互补关系，-2与10对模12互补，也可以说-2的补数是10(以12为模)。

可见，只要确定了“模”，就可找到一个与负数等价的正数(该正数即为负数的补数)来代替此负数，而这个正数可以用模加上负数本身求得，这样就可把减法运算用加法实现了。

例 2-3 $9-5=9+(-5)=9+(12-5)=9+7=4\ (\bmod 12)$

例 2-4 $65-25=65+(-25)=65+(100-25)=65+75=40\ (\bmod 100)$

将补数的概念用到计算机中，便出现了补码这种机器数。

$$[X]_{补}=\begin{cases}X & 0\leqslant X<\dfrac{M}{2}\\ M+X & -\dfrac{M}{2}\leqslant X<0\end{cases}\quad(\bmod M)$$

2. 补码表示

补码的符号位表示方法与原码相同，其数值部分的表示与数的正负有关：对于正数，数值部分与真值形式相同；对于负数，将真值的数值部分按位取反，且在最低位上加1。

若真值为纯小数,它的补码形式为 $X_s.X_1X_2\cdots X_n$,其中 X_s表示符号位。补码的定义为:

$$[X]_{补}=\begin{cases}X & 0\leqslant X<1\\ 2+X=2-|X| & -1\leqslant X<0\end{cases}\pmod 2$$

假设机器数长度为5位,则有:

例 2-5 $X=0.0110$ $[X]_{补}=X=0.0110$

$X=-0.0110$ $[X]_{补}=2+X=2+(-0.0110)=10-0.0110=1.1010$

若真值为纯整数,它的补码形式为 $X_sX_1X_2\cdots X_n$,其中 X_s 表示符号位。补码的定义为:

$$[X]_{补}=\begin{cases}X & 0\leqslant X<2^n\\ 2^{n+1}+X=2^{n+1}-|X| & -2^n\leqslant X<0\end{cases}\pmod{2^{n+1}}$$

例 2-6 $X=1101$ $[X]_{补}=X=01101$

$X=-1101$ $[X]_{补}=2^{n+1}+X=2^5+(-1101)=100000-1101=10011$

在补码表示中,真值0的表示形式是唯一的:

$$[+0]_{补}=[-0]_{补}=00000$$

3. 由真值、原码转换为补码

采用补码系统的计算机需要将真值或原码形式表示的数据转换为补码形式,以便于运算器对其进行运算。通常,从原码形式入手来求补码。

当 X 为正数时,$[X]_{补}=[X]_{原}=X$。

当 X 为负数时,其$[X]_{补}$等于把$[X]_{原}$除去符号位外的各位求反后最低位再加1。

反之,当 X 为负数时,已知$[X]_{补}$,也可通过对其除符号位外的各位求反加1来求得$[X]_{原}$。

当 X 为负数时,由$[X]_{原}$转换为$[X]_{补}$的另一种更有效的方法是:自低位向高位,尾数的第一个1及其右部的0保持不变,左部的各位取反,符号位保持不变。

例 2-7 $[X]_{原}=1.1110011000$

$[X]_{补}=\underline{1}.\underline{000110}\ \underline{1000}$

不变 取反 不变

这种方法避免了加1运算,是实际求补线路逻辑实现的依据。

对于负数,也可以直接由真值 X 转换为$[X]_{补}$,其方法更简单:数值位自低位向高位,尾数的第一个1及其右部的0保持不变,左部的各位取反,负号用1表示。

例 2-8 $X=-0.1010001010$

$[X]_{补}=1.0101110110$

2.1.5 反码表示法

反码表示法与补码表示法有许多类似之处,对于正数,数值部分与真值形式相同;对于负数,将真值的数值部分按位取反。反码与补码的区别是末位少加一个1,因此很容易从补码的定义推出反码的定义。

若真值为纯小数,它的反码形式为 $X_s.X_1X_2\cdots X_n$,其中 X_s表示符号位。反码的定义为:

$$[X]_{反}=\begin{cases}X & 0\leqslant X<1 \\ (2-2^{-n})+X & -1<X\leqslant 0\end{cases}\quad (\mathrm{mod}\ 2-2^{-n})$$

假设机器数长度为 5 位，则有：

例 2-9　$X=0.0110$　　$[X]_{反}=X=0.0110$

$X=-0.0110$　$[X]_{反}=(2-2^{-n})+X=(2-2^{-4})+X$

$=1.1111+(-0.0110)=1.1111-0.0110=1.1001$

若真值为纯整数，它的反码形式为 $X_sX_1X_2\cdots X_n$，其中 X_s 表示符号位。反码的定义为：

$$[X]_{反}=\begin{cases}X & 0\leqslant X<2^n \\ (2^{n+1}-1)+X & -2^n<X\leqslant 0\end{cases}\quad (\mathrm{mod}\ 2^{n+1}-1)$$

例 2-10　$X=1101$　　$[X]_{反}=X=01101$

$X=-1101$　$[X]_{反}=(2^{n+1}-1)+X=(2^5-1)+(-1101)$

$=11111-1101=10010$

在反码表示中，真值 0 也有两种不同的表示形式：

$$[+0]_{反}=00000$$

$$[-0]_{反}=11111$$

2.1.6　3 种机器数的比较与转换

1. 比较

原码、补码和反码这 3 种机器数既有共同点，又有各自不同的性质，主要区别有以下几点：

① 对于正数它们都等于真值本身，对于负数则各有不同的表示。

② 最高位都表示符号位，补码和反码的符号位可作为数值位的一部分看待，和数值位一起参加运算；但原码的符号位不允许和数值位同等看待，必须分开进行处理。

③ 对于真值 0，原码和反码各有两种不同的表示形式，而补码只有唯一的一种表示形式。

④ 原码、反码表示的正、负数范围相对零来说是对称的；但补码负数表示范围较正数表示范围宽，能多表示一个最负的数(绝对值最大的负数)，其值等于 -2^n(纯整数)或 -1(纯小数)。

表 2-1 列出了真值与 3 种机器数间的对照，表中设字长等于 4 位(含 1 位符号位)。

表 2-1　真值与 3 种机器数间的对照

真　值　X		$[X]_{原}$、$[X]_{反}$、$[X]_{补}$	真　值　X		$[X]_{原}$	$[X]_{反}$	$[X]_{补}$
十进制	二进制		十进制	二进制			
+0	+000	0000	−0	−000	**1000**	1111	0000
+1	+001	0001	−1	−001	1001	1110	1111
+2	+010	0010	−2	−010	1010	1101	1110
+3	+011	0011	−3	−011	1011	1100	1101
+4	+100	0100	−4	−100	1100	1011	1100
+5	+101	0101	−5	−101	1101	1010	1011
+6	+110	0110	−6	−110	1110	1001	1010
+7	+111	0111	−7	−111	1111	**1000**	1001
+8	—	—	−8	−1000	—	—	**1000**

在表2-1中,请特别注意1000这个代码,当其为原码时,对应的真值是−0;当其为补码时,对应的真值是−8;当其为反码时,对应的真值是−7。

2. 转换

3种不同机器数以及真值之间的转换关系如图2-1所示。

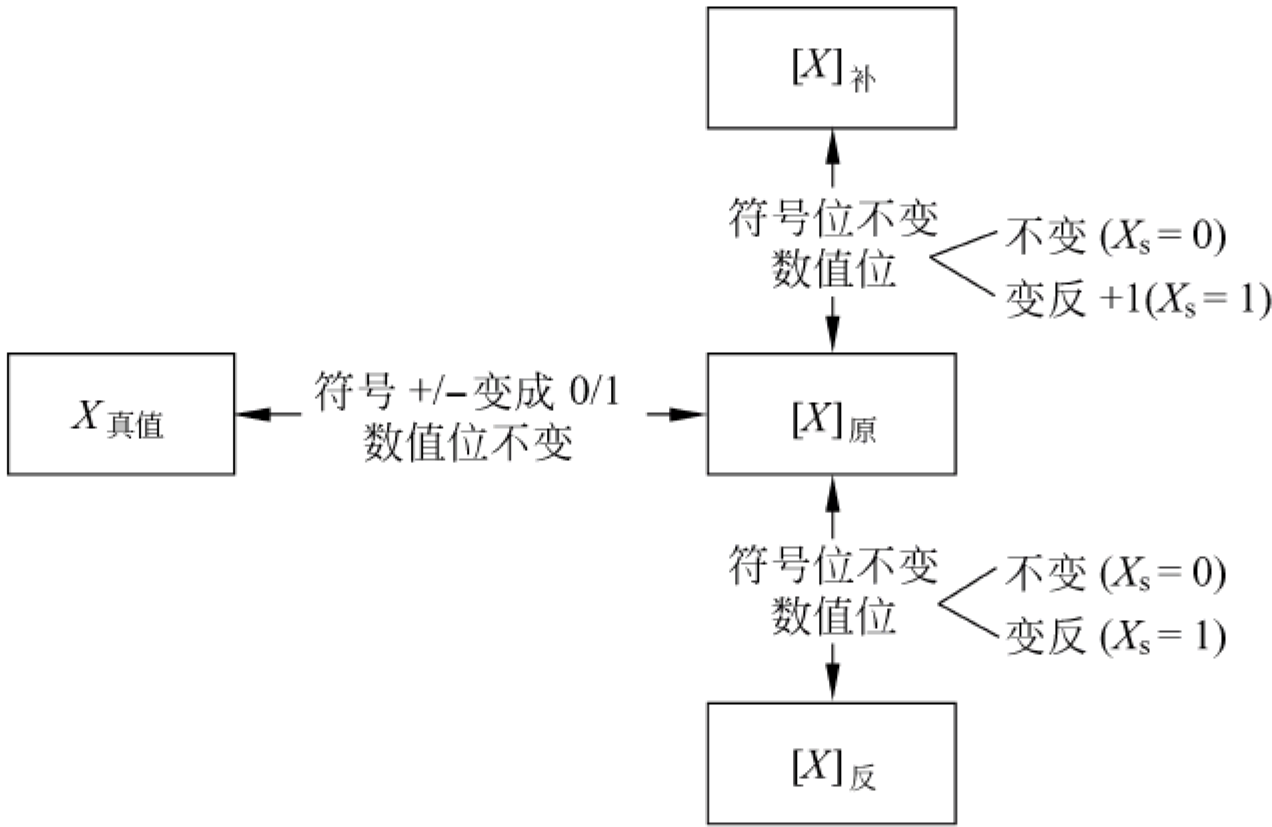


图2-1　3种不同机器数及真值间的转换关系

从图2-1可看出,真值 X 与补码或反码之间的转换通常是通过原码实现的,对于已熟练掌握转换方法的读者,也可以直接完成真值与补码或反码之间的转换。

如果已知机器的字长,则机器数的位数应补够相应的位数。例如,设机器字长为8位,则:

$X=1011$	$[X]_{原}=00001011$	$[X]_{补}=00001011$	$[X]_{反}=00001011$
$X=-1011$	$[X]_{原}=10001011$	$[X]_{补}=11110101$	$[X]_{反}=11110100$
$X=0.1011$	$[X]_{原}=0.1011000$	$[X]_{补}=0.1011000$	$[X]_{反}=0.1011000$
$X=-0.1011$	$[X]_{原}=1.1011000$	$[X]_{补}=1.0101000$	$[X]_{反}=1.0100111$

2.2　机器数的定点表示与浮点表示

计算机在进行算术运算时,需要指出小数点的位置。根据小数点的位置是否固定,在计算机中有两种数据格式:定点表示和浮点表示。

2.2.1　定点表示法

在定点表示法中约定:所有数据的小数点位置固定不变。通常,把小数点固定在有效数位的最前面或末尾,这就形成了两类定点数。

1. 定点小数

定点小数即纯小数,小数点的位置固定在最高有效数位之前、符号位之后,如图2-2所示。定点小数的小数点位置是隐含约定的,小数点并不需要真正地占据一个二进制位。

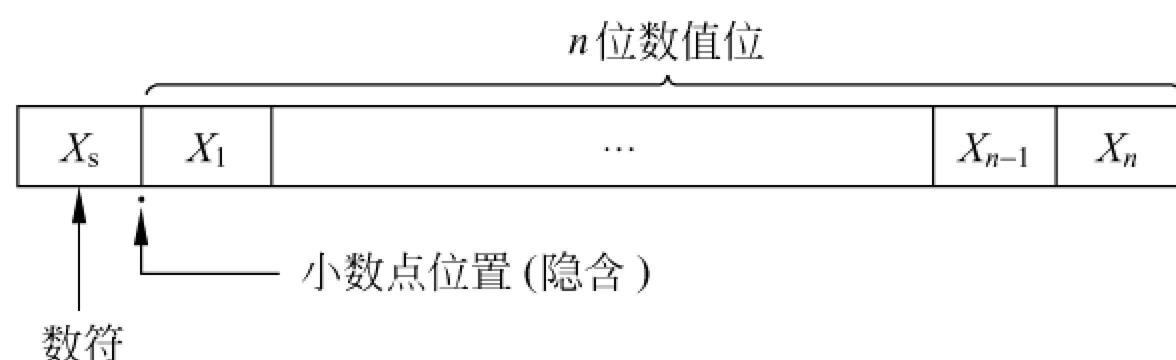


图 2-2　定点小数格式

当 $X_s=0, X_1=1, X_2=1, \cdots, X_n=1$ 时，X 为最大正数，其真值为：

$$X_{最大正数}=1-2^{-n}$$

当 $X_s=0, X_1=0, \cdots, X_{n-1}=0, X_n=1$ 时，X 为最小正数，其真值为：

$$X_{最小正数}=2^{-n}$$

当 $X_s=1$，表示 X 为负数时，情况要稍微复杂一些，这是因为在计算机中带符号数可用补码表示，也可用原码表示，原码和补码的表示范围有一些差别。

若机器数为原码，当 $X_s \sim X_n$ 均等于 1 时，X 为绝对值最大的负数，其真值为：

$$X_{绝对值最大负数}=-(1-2^{-n})$$

若机器数为补码，当 $X_s=1, X_1 \sim X_n$ 均等于 0 时，X 为绝对值最大的负数，其真值为：

$$X_{绝对值最大负数}=-1$$

综上所述，设机器字长有 $n+1$ 位，原码定点小数的表示范围为 $-(1-2^{-n}) \sim (1-2^{-n})$，补码定点小数的表示范围为 $-1 \sim (1-2^{-n})$。若字长为 8 位，原码定点小数的表示范围为 $-\frac{127}{128} \sim \frac{127}{128}$，补码定点小数的表示范围为 $-1 \sim \frac{127}{128}$。

2. 定点整数

定点整数即纯整数，小数点位置隐含固定在最低有效数位之后，如图 2-3 所示。



图 2-3　定点整数格式

根据前述方法不难推出：

$$X_{最大正数}=(2^n-1)$$

$$X_{最小正数}=1$$

$$X_{绝对值最大负数}=-(2^n-1)（原码表示时）$$

$$X_{绝对值最大负数}=-2^n（补码表示时）$$

综上所述，设机器字长为 $n+1$ 位，原码定点整数的表示范围为 $-(2^n-1) \sim (2^n-1)$，补码定点整数的表示范围为 $-2^n \sim (2^n-1)$。若字长为 8 位，原码定点整数的表示范围为 $-127 \sim 127$，补码定点整数的表示范围为 $-128 \sim 127$。

在定点表示法中,参加运算的数以及运算的结果都必须保证落在该定点数所能表示的数值范围内,如结果大于最大正数和小于绝对值最大的负数,统称为"溢出"。这时计算机将暂时中止运算操作,而进行溢出处理。

需要说明的是,现代计算机中大多只采用整数数据表示,小数则通过浮点数表示来实现。

2.2.2 浮点表示法

在科学计算中,计算机处理的数往往是混合数,它既有整数部分又有小数部分,如果要将这些数变为上述约定的两种定点数形式,就必须在运算前设定一个比例因子,把原始的数缩小成定点小数或扩大成定点整数,运算后的结果还需要根据比例因子还原成实际的数值,这会给编程带来很多麻烦。另外,在运算中常常会遇到非常大或非常小的数值,如果用同样的比例因子来处理,很难兼顾数值范围和运算精度的要求。因此,在计算机中引入了浮点数表示。

让小数点的位置根据需要而浮动,这就是浮点数。例如:

$$N=M\times r^E$$

其中,r 是浮点数阶码的底,与尾数的基数相同,通常 $r=2$。E 和 M 都是带符号的定点数,E 称为阶码(Exponent),M 称为尾数(Mantissa)。在大多数计算机中,尾数为纯小数,常用原码或补码表示;阶码为纯整数,常用移码或补码表示。

浮点数的一般格式如图 2-4 所示,浮点数的底是隐含的,在整个机器数中不出现。阶码的符号位为 e_s,阶码的大小反映了在数 N 中小数点的实际位置;尾数的符号位为 m_s,它也是整个浮点数的符号位,表示了该浮点数的正、负。

浮点数的表示范围主要由阶码的位数来决定,有效数字的精度主要由尾数的位数来决定,下面介绍这两个问题。

1. 浮点数的表示范围

设某浮点数的格式如图 2-4 所示,k 和 n 分别表示阶码和尾数的数值位位数(不包括符号位),尾数和阶码均用补码表示。

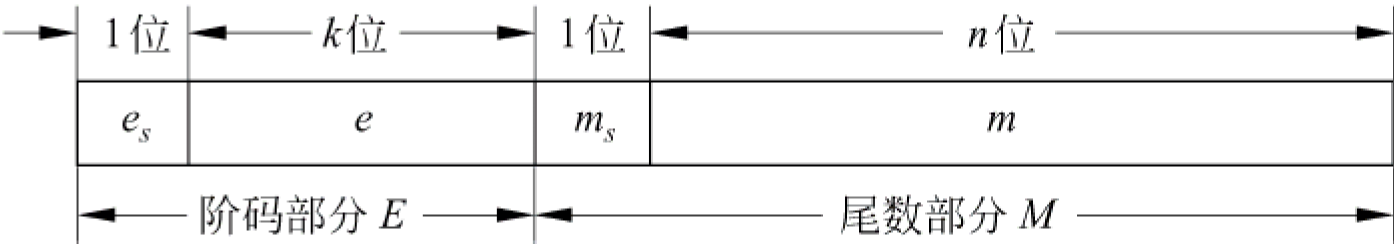


图 2-4 浮点数的一般格式

当 $e_s=0$,$m_s=0$,阶码和尾数的数值位各位全为 1(即阶码和尾数都为最大正数)时,该浮点数为最大正数:

$$X_{\text{最大正数}}=(1-2^{-n})\times 2^{2^k-1}$$

当 $e_s=1$,$m_s=0$,尾数的最低位 $m_n=1$,其余各位为 0(即阶码为绝对值最大的负数,尾数为最小正数)时,该浮点数为最小正数:

$$X_{\text{最小正数}}=2^{-n}\times 2^{-2^k}$$

当 $e_s=0$,阶码的数值位为全 1;且 $m_s=1$,尾数的数值位为全 0(即阶码为最大正数,尾

数为绝对值最大的负数)时,该浮点数为绝对值最大负数:

$$X_{绝对值最大负数} = -1\times 2^{2^k-1}$$

2. 规格化浮点数

为了提高运算的精度,需要充分地利用尾数的有效数位,通常采取浮点数规格化形式,即规定尾数的最高数位必须是一个有效值。

一个浮点数的表示形式并不是唯一的。例如,二进制数 0.0001101 可以表示为 0.001101×2^{-01}、0.01101×2^{-10}、0.1101×2^{-11}……而其中只有 0.1101×2^{-11} 是规格化数,这就如同十进制实数中的科学标识法一样。

规格化浮点数的尾数 M 的绝对值应在下列范围内:

$$\frac{1}{r}\leqslant |M|<1$$

如果 $r=2$,则有$\frac{1}{2}\leqslant |M|<1$。在尾数用原码表示时,规格化浮点数的尾数的最高数位总等于 1。在尾数用补码表示时,规格化浮点数应满足尾数最高数位与符号位不同($m_s\oplus m_1=1$),即当$\frac{1}{2}\leqslant M<1$ 时,应有 $0.1\times\times\cdots\times$ 形式;当 $-1\leqslant M<-\frac{1}{2}$ 时,应有 $1.0\times\times\cdots\times$形式。

注意:当 $M=-\frac{1}{2}$时,对于原码来说这是一个规格化数,而对于补码来说这不是一个规格化数;当 $M=-1$ 时,对于原码来说这将无法表示,而对于补码来说这是一个规格化数。

当 $e_s=1$,$m_s=0$,尾数的最高位 $m_1=1$,其余各位为 0 时,该浮点数为规格化的最小正数:

$$X_{规格化的最小正数}=2^{-1}\times 2^{-2^k}$$

规格化的最小正数大于非规格化的最小正数。

非规格化浮点数需要进行规格化操作才能变成规格化浮点数。所谓规格化操作,就是通过相应地调整一个非规格化浮点数的尾数和阶码的大小,使非零的浮点数在尾数的最高数位上保证是一个有效值,具体的操作方法留待第 4 章介绍。

表 2-2 列出了浮点数的一些典型值,设阶码和尾数均用补码表示,阶码共 $k+1$ 位(含一位阶符),尾数共 $n+1$ 位(含一位尾符)。

表 2-2 浮点数的典型值

浮点数	浮点数代码		真值
	阶码	尾数	
最大正数	01…1	0.11…11	$(1-2^{-n})\times 2^{2^k-1}$
绝对值最大负数	01…1	1.00…00	$-1\times 2^{2^k-1}$
最小正数	10…0	0.00…01	$2^{-n}\times 2^{-2^k}$
规格化的最小正数	10…0	0.10…00	$2^{-1}\times 2^{-2^k}$
绝对值最小负数	10…0	1.11…11	$-2^{-n}\times 2^{-2^k}$
规格化的绝对值最小负数	10…0	1.01…11	$-(2^{-1}+2^{-n})\times 2^{-2^k}$

当运算结果大于最大正数时称为正上溢,小于绝对值最大负数时称为负上溢,正上溢和负上溢统一称为上溢,数据一旦产生上溢,计算机必须中止运算操作,进行溢出处理。当运算结果在0至规格化最小正数之间称为正下溢,在0至规格化的绝对值最小负数之间称为负下溢,正下溢和负下溢统一称为下溢,数据一旦出现下溢,计算机一般不做任何处理,仅仅置成机器零即可。

只要浮点数的尾数为0,不论阶码为何值,一般也当作机器零处理。为了保证浮点数0表示形式的唯一性,规定了机器零的标准格式,即:尾数为0,阶码为最小值(绝对值最大的负数)。

2.2.3 浮点数阶码的移码表示法

浮点数的阶码是带符号的定点整数,理论上说它可以用前面提到的任何一种机器数的表示方法来表示,但在多数通用计算机中,它还采用另一种编码方法——移码表示法。

移码就是在真值 X 基础上加一个常数,这个常数被称为偏置值,相当于 X 在数轴上向正方向偏移了若干单位,这就是"移码"一词的由来,移码也可称为增码或偏码。即:

$$[X]_{移}=偏置值+X$$

图2-5是移码和真值的映射图,此时偏置值等于 2^n。

对于字长为8位的定点整数,如果偏置值为 2^7,则有:

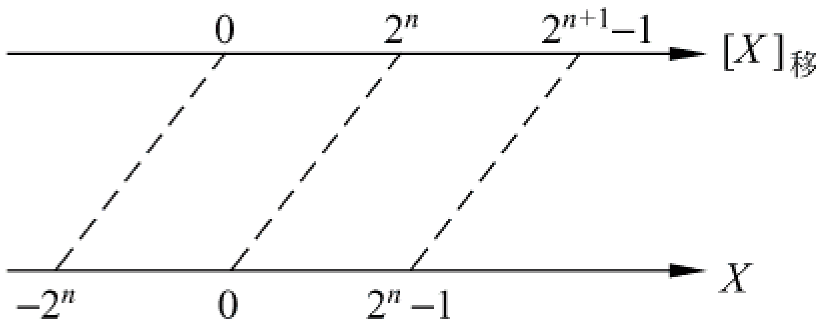


图2-5 移码和真值的映射

例 2-11 $X=1101101$

$[X]_{移}=2^7+1101101=10000000+1101101$

$=\mathbf{1}1101101$

而此时 $[X]_{补}=\mathbf{0}1101101$。

例 2-12 $X=-1101101$

$[X]_{移}=2^7+(-1101101)=10000000-1101101=\mathbf{0}0010011$

而此时 $[X]_{补}=\mathbf{1}0010011$。

表2-3给出了偏置值为 2^7 的移码与补码和真值之间的关系。

表 2-3 偏置值为 2^7 的移码、补码和真值之间的关系

真值 X(十进制)	真值 X(二进制)	$[X]_{补}$	$[X]_{移}$
−128	−10000000	10000000	00000000
−127	−1111111	10000001	00000001
⋮	⋮	⋮	⋮
−1	−0000001	11111111	01111111
0	0000000	00000000	10000000
1	0000001	00000001	10000001
⋮	⋮	⋮	⋮
127	1111111	01111111	11111111

从表2-3中可以看出这种移码具有以下特点:

① 在移码中,最高位为0表示负数,最高位为1表示正数,这与原码、补码以及反码的符号位取值正好相反。

② 移码全为0时，它所对应的真值最小；全为1时，它所对应的真值最大。因此，移码的大小直观地反映了真值的大小，这将有助于两个浮点数进行阶码的大小比较。

③ 真值0在移码中的表示形式也是唯一的，即$[+0]_{移}=[-0]_{移}=10000000$。

④ 移码把真值映射到一个正数域，所以可将移码视为无符号数，直接按无符号数规则比较大小。

⑤ 同一数值的移码和补码除最高位相反外，其他各位相同。

浮点数的阶码常采用移码表示的最主要的原因有两个：

① 便于比较浮点数的大小。阶码大的，其对应的真值就大；阶码小的，则对应的真值就小。

② 简化机器中的判零电路。当阶码全为0，尾数也全为0时，表示机器零。

2.2.4 浮点数尾数的基数

1. 尾数基数大小的选择

已知，浮点数$N=M\times r^E$。其中，r是阶码的底，又称为尾数的基数，在前面的讨论中，r都等于2，而实际上r可以等于4,8,16……

浮点数尾数基数的选择对浮点数的特性起着主要作用，它既影响浮点运算的精度，也影响数值的表示范围。采用较大的r值，在阶码位数相同的情况下，可以扩大浮点数的表示范围。假定某浮点数字长为32位，阶码部分(阶符和阶码数值位)共8位，尾数部分(尾符与尾数数值位)共24位，均用补码表示。

若$r=2$，则规格化浮点数的表示范围为$-1\times 2^{127}\leqslant X\leqslant(1-2^{-23})\times 2^{127}$，最小正数为$2^{-1}\times 2^{-128}$。

若$r=16$，则规格化浮点数的表示范围为$-1\times 16^{127}\leqslant X\leqslant(1-2^{-23})\times 16^{127}$，最小正数为$16^{-1}\times 16^{-128}$。

事实上，在以r进制为尾数基数的浮点数中，当尾数的位数为二进制n位时，就相当于r进制的尾数有n'个数位，其中$n'=n/\lceil\log_2 r\rceil$。

前面已经提到，规格化也是和基数r有关系的，规格化浮点数的尾数$|M|$应在$\left(\frac{1}{r},1\right)$范围内。若现在尾数$M=0.0001\times\cdots\times$，对于基数$r=2$的浮点数来说，这是一个非规格化数，需要进行规格化操作；而对于$r=16$的浮点数来说，这已是一个规格化数了，无须再进行规格化操作。规格化操作是以r为尺度进行移位的，尾数每左移(或右移)$\lceil\log_2 r\rceil$位，阶码将减(或加)1。

不同计算机的r值可能不同。例如，PDP-11和IBM 370的短浮点数具有同样的格式，但前者$r=2$，后者$r=16$。所以，IBM 370的短浮点数比PDP-11的短浮点数的表示范围要大，但相对误差也较大。

2. 尾数基数r对浮点数特性的影响

改变尾数基数r，会使浮点数的特性有下列影响：

① 可表示数的范围。随着r的增大，可表示数的范围增大。

② 可表示数的个数。随着 r 的增大,可表示数的个数增加。

③ 数在数轴上的分布。r 愈大,数在数轴上的分布密度愈稀。

④ 可表示数的精度。随着 r 的增大,可表示数的精度单调下降。

⑤ 运算中的精度损失。r 愈大,尾数右移的机会愈小,可降低运算中因尾数右移所造成的精度损失。

⑥ 运算速度。r 增大,将使运算中移位的次数减少,运算速度可以提高。

以上6条中,第①、②、⑤、⑥条都是优点;但第③、④条则是缺点。为了扬长避短,在巨型、大型和中型计算机上,浮点数尾数基数 r 宜取大些,而在微型和小型计算机上,r 值宜取小些。

2.2.5 IEEE 754 标准浮点数

在目前常用的80x86系列微型计算机中,通常设有支持浮点运算的部件。在这些机器中的浮点数采用IEEE 754标准,它与前面介绍的浮点数格式有一些差别。

按IEEE 754标准,常用的浮点数的格式如图2-6所示。

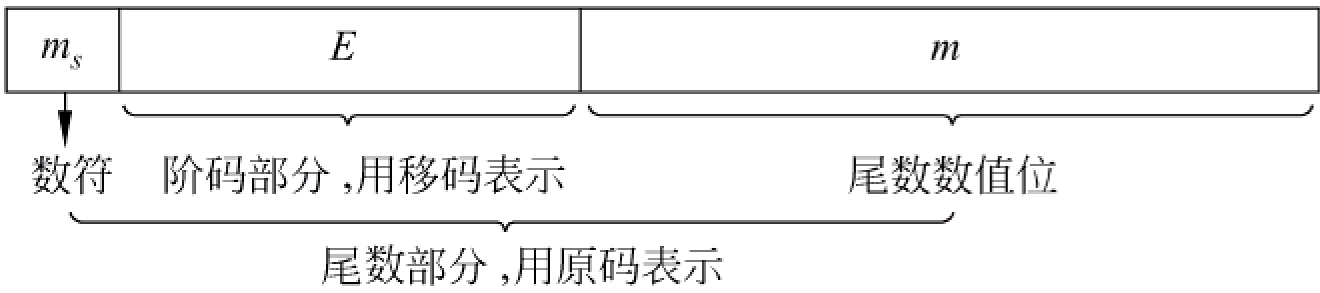


图2-6 按IEEE 754标准,常用的浮点数格式

IEEE 754标准中有3种形式的浮点数,它们的具体格式如表2-4所示。

表2-4 IEEE 754标准中的3种浮点数

类型	数符	阶码	尾数数值	总位数	偏置值	
					十六进制	十进制
短浮点数	1	8	23	32	7FH	127
长浮点数	1	11	52	64	3FFH	1023
临时浮点数	1	15	64	80	3FFFH	16383

短浮点数又称为单精度浮点数,长浮点数又称为双精度浮点数,它们都采用隐含尾数最高数位的方法,这样无形中又增加了一位尾数。临时浮点数又称为扩展精度浮点数,它没有隐含位。

下面以32位的短浮点数为例,讨论浮点代码与其真值之间的关系。最高位为数符位;其后是8位阶码,以2为底,用移码表示,阶码的偏置值为127;其余23位是尾数数值位。对于规格化的二进制浮点数,数值的最高位总是1,为了能使尾数多表示一位有效值,可将这个1隐含,因此尾数数值实际上是24位(1位隐含位+23位小数位)。

注意:隐含的1是一位整数(即位权为 2^0)。在浮点格式中表示出来的23位尾数是纯小数,用原码表示。例如,$(12)_{10}=(1100)_2$,将它规格化后结果为 1.1×2^3,其中整数部分的"1"将不存储在23位尾数内。

阶码是以移码形式存储的。对于短浮点数,偏置值为127(7FH);长浮点数,偏置值为

1023(3FFH)。存储浮点数阶码部分之前,偏置值要先加到阶码真值上。上述例子中,阶码真值为3,故在短浮点数中,移码表示的阶码为127+3=130(82H);而在长浮点数中,移码表示的阶码为1023+3=1026(402H)。

例 2-13 将$(100.25)_{10}$转换成短浮点数格式。

① 把十进制数转换成二进制数。

$$(100.25)_{10}=(1100100.01)_2$$

② 规格化二进制数。

$$1100100.01=1.10010001\times 2^6$$

③ 计算出阶码的移码(偏置值+阶码真值)。

$$1111111+110=10000101$$

④ 以短浮点数格式存储该数。

因为,

符号位=0
阶码=10000101
尾数=10010001000000000000000

所以,短浮点数代码为:

0;10000101;10010001000000000000000

表示为十六进制的代码:42C88000H。

例 2-14 把短浮点数C1C90000H转换成十进制数。

① 将十六进制代码写成二进制形式,并分离出符号位、阶码和尾数。

因为,

C1C90000H=11000001110010010000000000000000

所以,

符号位=1
阶码=10000011
尾数=10010010000000000000000

② 计算出阶码真值(移码减去偏置值)。

$$10000011-1111111=100$$

③ 以规格化二进制数形式写出此数。

$$1.1001001\times 2^4$$

④ 写成非规格化二进制数形式。

$$11001.001$$

⑤ 转换成十进制数,并加上符号位。

$$(11001.001)_2=(25.125)_{10}$$

所以,

该浮点数=-25.125

通常,将IEEE 754短浮点数规格化的数值v表示为:

$$v=(-1)^S\times(1.f)\times 2^{E-127}$$

其中,S代表符号位,$S=0$表示正数,$S=1$表示负数;E为用移码表示的阶码;f是尾数的小

数部分。

为了表示∞和一些特殊的数值,E 的最小值 0 和最大值 255 将留作他用。因此,最小正常的 $E=1$,最大正常的 $E=254$,所以短浮点数的阶码真值的取值范围为 $-126\sim127$。当 E 和 m 均为全 0 时,表示机器零;当 E 为全 1,m 为全 0 时,表示 $\pm\infty$。

2.2.6 定点、浮点表示法与定点、浮点计算机

1. 定点、浮点表示法的区别

(1) 数值的表示范围

假设定点数和浮点数的字长相同,浮点表示法所能表示的数值范围将远远大于定点表示法。

浮点数阶码部分的位数占得越多,可表示的数值范围就越大,但是相应尾数部分的位数将减少,这将使精度下降。因此,阶码和尾数部分各占多少位,必须全面权衡,合理分配。

注意:

- 不管定点数还是浮点数,每个数值都对应于数轴上的一个点。所谓数的表示范围,实际上指的只是数的上、下限,它们之间是一些不连续的点,而不是一段连续的区间。
- 对于定点整数而言,各个点在数轴上的分布是均匀的;而对于浮点数而言,各个点在数轴上的分布是不均匀的,越靠近数轴的原点,两个相邻数之间的距离就越近。

(2) 精度

所谓精度是指一个数所含有效数值位的位数。一般来说,机器字长越长,它所表示的数的有效位数就越多,精度就越高。对于字长相同的定点数与浮点数来说,浮点数虽然扩大了数的表示范围,但这正是以降低精度为代价的,也就是数轴上各点的排列更稀疏了。

(3) 数的运算

浮点数包括阶码和尾数两部分,运算时不仅要做尾数的运算,还要做阶码的运算,而且运算结果要求规格化。因此,浮点运算要比定点运算复杂,关于具体运算的讨论将在第 4 章中进行。

(4) 溢出处理

在定点运算时,当运算结果超出数的表示范围时,就发生溢出;而在浮点运算时,运算结果超出尾数的表示范围却并不一定溢出,只有当阶码也超出所能表示的范围时,才发生溢出。

2. 定点机与浮点机

由于浮点数的运算比较复杂,所以并不是所有的计算机都具有浮点运算功能,通常可以将计算机分为以下几种。

(1) 定点机

只能处理定点数的计算机称为定点计算机,低档微机和某些专用机大多是定点机。在定点机中机器指令访问的所有操作数都是定点数,如需进行浮点运算则通过执行软件子程序来实现。

(2) 定点机+浮点运算部件

浮点运算部件是专门用于对计算机内的浮点数进行运算的部件,系统配置了浮点运算

部件，将使浮点运算速度大大提高。许多微、小型计算机都配有这一部件。

(3) 浮点机

具有浮点运算指令和基本的浮点运算器。通用的大、中型计算机多为浮点机。

2.3 非数值数据的表示

非数值数据，又称为字符数据，通常是指字符、字符串、图形符号和汉字等各种数据，它们不用来表示数值的大小，一般情况下不对它们进行算术运算。

2.3.1 字符和字符串的表示

1. ASCII 字符编码

由于计算机内部只能识别和处理二进制代码，所以字符必须按照一定的规则用一组二进制编码来表示。字符编码方式有很多种，现在用得最广泛的是美国国家信息交换标准字符码(American Standard Code for Information Interchange，ASCII)。

常见的 ASCII 码用 7 位二进制表示一个字符，ASCII 码包括 10 个十进制数字(0～9)、52 个英文大写和小写字母(A～Z 和 a～z)、34 个专用符号和 32 个控制符号，共计 128 个字符。在 128 个字符中有 96 个是可打印字符。

在计算机中，通常用一个字节来存放一个字符。对于 ASCII 码来说，一个字节右边的 7 位表示不同的字符代码，而最左边一位可以作为奇偶校验位，用来检查错误，也可以用于西文字符和汉字的区分标识。

ASCII 字符编码表如表 2-5 所示。由表中可见，数字和英文字母都是按顺序排列的，只要知道其中一个的二进制代码，不用查表就可以推导出其他数字或字母的二进制代码。另外，如果将 ASCII 码中 0～9 十个数字的二进制代码去掉最高 3 位 011，正好与它们的二进制值相同，这不但使十进制数字进入计算机后易于压缩成 4 位代码，而且也便于进一步的信息处理。

表 2-5　ASCII 字符编码表

$b_3b_2b_1b_0$ \ $b_6b_5b_4$	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z

续表

$b_6b_5b_4$ / $b_3b_2b_1b_0$	000	001	010	011	100	101	110	111
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	−	=	M	]	m	}
1110	RO	RS	.	>	N	↑	n	~
1111	SI	US	/	?	O	_	o	DEL

除标准ASCII字符编码外,许多公司还使用8位二进制编码来表示更大的字符集,例如IBM公司就用8位扩展二进制编码的十进制交换码(EBCDIC码)来表示IBM计算机所用到的字符集。

2. 字符串的存放

字符串是指一串连续的字符。通常,它们在存储器中占用一片连续的空间,每个字节存放一个字符代码,字符串的所有元素(字符)在物理上是邻接的,这种字符串的存储方法称为向量法。例如,字符串IF X>0 THEN READ (C),在字长为32位的存储器中的存放格式如图2-7(a)所示。图中每一个主存单元可存放4个字符,整个字符串需5个主存单元。在每个字节中实际存放的是相应字符的ASCII码,如图2-7(b)所示。

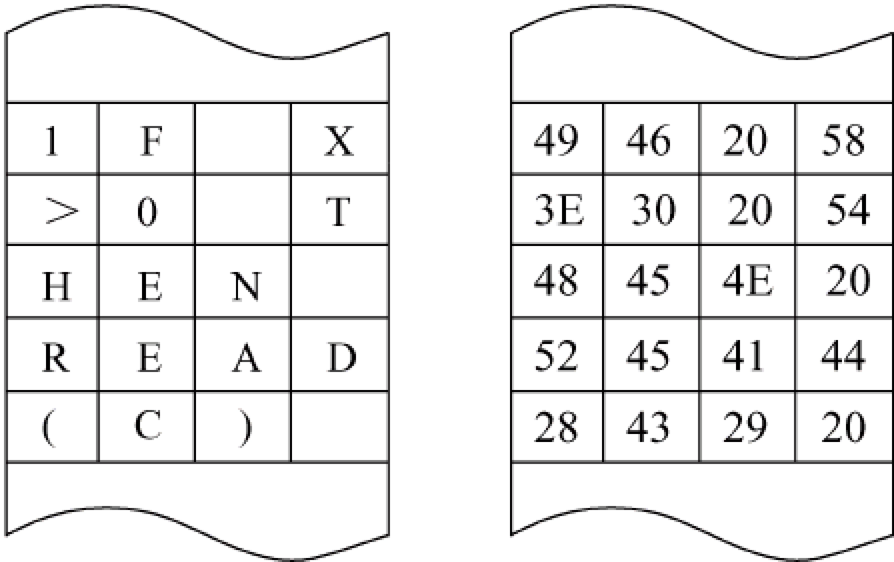


图2-7 字符串的向量存放方案

字符串的向量存放法是最简单、最节省存储空间的方法。但是,当字符串需要进行删除和插入操作时,在删除或插入字符后面的子字符串需要全部重新分配存储空间,将花费较多的时间。为了克服向量存放法的缺点,另一种字符串的存储方法——串表法应运而生了。在这种存储方法中,字符串的每个字符代码后有一个链接字,用以指出下一个字符的存储单元地址。串表法不要求串中的各个字符在物理上相邻,原则上讲,串中各字符可以安排在存储器的任意位置上。在对字符串进行删除和插入操作时,只需修改相应字符代码后面的链接字即可,所以非常方便。但是,由于链接字占据了存储单元的大部分空间,使得主存的有效利用率下降。例如,一个主存单元有32位,仅存放一个字符代码,而链接字占用了24位,这时,存放字符串信息的主存有效利用率只占25%,这是串表法的最大缺点。

2.3.2 汉字的表示

汉字处理技术是计算机推广应用工作中必须要解决的问题。汉字的字数繁多,字形复杂,读音多变,常用的汉字就有7000个左右。要在计算机中表示汉字,最方便的方法是为汉字安排一个编码,而且要使这些编码与西文字符和其他字符有明显的区别。

1. 汉字国标码

汉字国标码也称为汉字交换码，主要用于汉字信息处理系统之间或者通信系统之间交换信息使用。1981年国家标准总局公布了GB 2312—80，即《信息交换用汉字编码字符集基本集》，简称GB码。该标准共收集常用汉字6763个，其中一级汉字3755个，按拼音排序。二级汉字3008个，按部首排序。另外还有各种图形符号682个，共计7445个。

GB 2312—80规定每个汉字、图形符号都用两个字节表示，每个字节只使用低7位编码，因此最多能表示出128×128=16 384个汉字。

2. 汉字区位码

区位码将汉字编码GB 2312—80中的6763个汉字分为94个区，每个区中包含94个汉字(位)，区和位组成一个二维数组，每个汉字在数组中对应一个唯一的区位码。汉字的区位码定长4位，前两位表示区号，后两位表示位号，区号和位号用十进制数表示，区号从01到94，位号也从01到94。例如，“中”字在54区的48位上，其区位码为“54－48”，“国”字在25区的90位上，其区位码为“25－90”。

区位码表的布局是这样安排的，第1～15区包含西文字母、数字和图形符号，以及用户自行定义的专用符号(统称非汉字图形字符)；第16～55区为一级汉字；第56～87区为二级汉字；87区以上为空白区，可供造新字用。

注意：汉字区位码并不等于汉字国标码，它们两者之间的关系可用以下公式表示：

国标码＝区位码(十六进制)＋2020H

即首先将十进制表示的区位码转换成十六进制表示，然后再加上2020H。

例 2-15　已知汉字“春”的区位码为“20—26”，计算它的国标码。

	第一字节	第二字节	
区位码：	20	26	十进制
	↓	↓	
	14H	1AH	十六进制
	＋ 20H	＋ 20H	
国标码：	34H	3AH	

使用区位码输入汉字时，每输入4位数字可得到一个汉字，没有重码，但由于要查阅区位码表，所以较麻烦。

3. 汉字机内码

汉字可以通过不同的输入码输入，但在计算机内部其内码是唯一的。因为汉字处理系统要保证中西文的兼容，当系统中同时存在ASCII码和汉字国标码时，将会产生二义性。例如，有两个字节的内容为30H和21H，它既可表示汉字“啊”的国标码，又可表示西文“0”和“!”的ASCII码。为此，汉字机内码应对国标码加以适当处理和变换。

汉字机内码也是两字节长的代码，它是在相应国标码的每个字节最高位上加1所得，即

汉字机内码＝汉字国标码＋8080H

例如，上述“啊”字的国标码是3021H，其机内码则是B0A1H。

4. 汉字字形码

汉字字形码是指确定一个汉字字形点阵的代码，又称汉字字模码或汉字输出码。在一个汉字点阵中，凡笔画所到之处，记为1，否则记为0。

根据对汉字质量的不同要求，可有16×16、24×24、32×32或48×48的点阵结构。汉字点阵分类如表2-6所示，显然点阵越大，输出汉字的质量越高，每个汉字所占用的字节数也越高。

表2-6 汉字点阵分类

字 形	点阵(行×列)	字节数	特 征
简易型	16×16	32	显示字体骨架
普及型	24×24	72	有笔锋，可分字体
提高型	32×32	128	笔锋清晰，字体齐全
精密型	48×48	288	能表示复杂字形

汉字字形码在汉字输出时经常使用，所以要把各个汉字的字形信息固定存储起来，存放各个汉字字形信息的实体称为汉字库。汉字库的信息量很大，一个16×16点阵的基本汉字库至少需要256KB，而24×24点阵的汉字库至少需576KB。

5. 汉字编码的发展

GB 2312是在中国大陆及海外使用简体中文的国家和地区(如新加坡等)强制使用的唯一中文编码，但它只有6000多个汉字，已不能满足各方面应用的需要。国家标准总局在1990年颁布了繁体字的编码标准GB 12345—90《信息交换用汉字编码字符集 第一辅助集》，目的在于规范必须使用繁体字的各种场合，该标准共收录6866个汉字(比GB 2312多103个字)，纯繁体的字大概有2200个，每个汉字都采用双字节编码。1995年年底推出的GBK编码是中文编码扩展国家标准，该编码标准兼容GB 2312，共收录汉字21 003个、符号883个，并提供1894个造字码位，简、繁体字融于一库，GBK也采用双字节编码。

随着国际间交流与合作的扩大，信息处理应用对字符集提出了多文种、大字量、多用途的要求。1993年国际标准化组织发布了ISO/ 10646—1《信息技术通用多八位编码字符集 第一部分体系结构与基本多文种平面》。我国根据此标准制定了GB 13000.1—93，该标准采用了全新的多文种编码体系，收录了中、日、韩(CJK统一汉字)20 902个，是编码体系未来的发展方向。由于其新的编码体系与现有多数操作系统和外部设备不兼容，所以它的实现仍需要一个过程。

考虑到GB 13000的完全实现有待时日，以及GB 2312编码体系的延续性和现有资源和系统的有效利用与过渡。2000年年底，我国国家标准总局颁布了GB 18030大字符集标准，这个标准全面兼容GB 2312，在字汇上兼容GB 13000。目前，GB 18030有两个版本：GB 18030—2000和GB 18030—2005。GB 18030—2000规定了常用非汉字符号和27 533个汉字(包括部首、部件等)的编码。GB 18030—2005是以汉字为主并包含多种我国少数民族文字(如藏、蒙古、傣、彝、朝鲜、维吾尔文等)的超大型中文编码字符集，其中收录汉字70 244个(在GB 18030—2000的基础上增加了42 711个)。GB 18030采用单字节、双字

节、4字节混合编码，编码空间超过150万个。

从ASCII、GB 2312、GBK到GB 18030，这些编码方法是向上兼容的，即同一个字符在这些方案中总是有相同的编码，后面的标准支持更多的字符。为了保证中西文兼容，确定中文编码的方法是：高字节的最高位为1，不用管低字节最高位是什么。

2.3.3 统一代码

统一代码（Unicode）又称万国码、单一码，它是由多语言软件制造商组成的统一码联盟制定的可以容纳世界上所有文字和符号的字符编码方案。Unicode随着通用字符集的标准而发展，至今仍在不断增加和修改，Unicode 1.0于1991年10月发布，截至目前为止已发布了24个版本，每个新版本都会加入更多新的字符，目前最新的版本为2018年6月发布的Unicode 11.0。

Unicode可分为编码方式和实现方式两个层次。

1. 编码方式

Unicode的编码方式与国际标准ISO/IEC 10646的通用字符集（Universal Character Set，UCS）概念相对应，两者的码表兼容，并共同调整任何未来的扩展。

Unicode的基本方法是用一个16位的二进制数（2字节）来表示Unicode中的每个符号，这意味着允许表示65 536个不同的字符或符号。这种符号集被称为基本多语言平面（BMP），基本多语言平面的字符编码为U+hhhh，其中每个h代表一个十六进制数字，与UCS-2编码完全相同。

UCS-4是一个更大的尚未填充完全的31位字符集，加上恒为0的首位，共需占据32位，即4字节。理论上最多能表示2^{31}个字符，完全可以涵盖一切语言所用的符号。目前，Unicode计划使用了17个平面（1个基本语言平面和16个辅助平面），一共有17×65 536=1 114 112个码位。

图2-8给出了在微机中用的扩展ASCII码、Unicode UCS-2和UCS-4方法表示一个字符之间的差异。例如，字符A用UCS-2表示为“U+0041”，而其对应的UCS-4的后两个字节与UCS-2一致，前两个字节则所有位均为0。

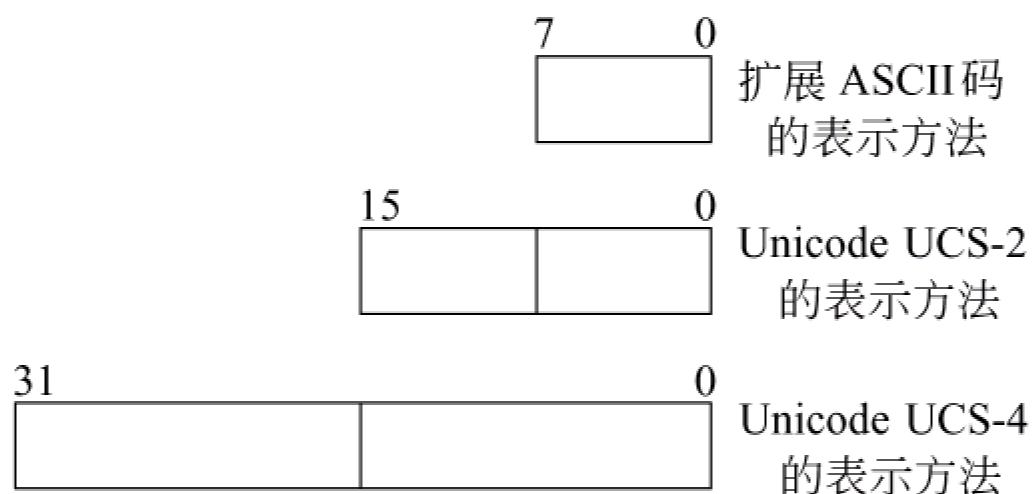


图2-8 微机中表示字符的3种方法

2. 实现方式

Unicode的实现方式不同于编码方式。一个字符的Unicode编码是确定的，但是在实际传输过程中，由于不同系统平台的设计不一定一致，以及出于节省空间的考虑，Unicode

编码的实现方式会有所不同。Unicode 的实现方式称为 Unicode 转换格式(Unicode Translation Format，UTF)，目前存在的 UTF 格式有 UTF-7、UTF-7.5、UTF-8、UTF-16 和 UTF-32。

例如，UTF-8 编码是一种变长编码，用 1～6 个字节编码 Unicode 字符。它将基本 7 位 ASCII 字符仍用 7 位编码表示，占用 1 个字节(首位补 0)。而遇到与其他 Unicode 字符混合的情况，将按一定算法转换，UCS-2 转换成 UTF-8 很可能需要 3 个字节，UCS-4 转换成 UTF-8 很可能需要 6 个字节。这样以 7 位 ASCII 字符为主的西文文档大大节省了编码长度。

UTF-16 比起 UTF-8，好处在于大部分字符都以固定长度的字节(2 字节)存储，但 UTF-16 却无法兼容 ASCII 编码。

2.4　十进制数和数串的表示

十进制是人们最常用的数据表示方法，一些通用性较强的计算机上设有十进制数据的表示，可以直接对十进制数进行运算和处理。

2.4.1　十进制数的编码

二进制是计算机最适合的数据表示方法，把十进制数的各位数字变成一组对应的二进制代码，用 4 位二进制数来表示 1 位十进制数，称为二进制编码的十进制数(Binary Code Decimal)，即 BCD 码。4 位二进制数可以组合出 16 种代码，能表示 16 种不同的状态，只需要使用其中的 10 种状态，就可以表示十进制数的 0～9 十个数码，而其他的 6 种状态为冗余状态。由于可以取任意的 10 种代码来表示 10 个数码，所以就可能产生多种 BCD 编码。BCD 编码既具有二进制数的形式，又保持了十进制数的特点，可以作为人机联系的一种中间表示，也可以用它直接进行运算。表 2-7 列出了几种常见的 BCD 码。

表 2-7　常见的 BCD 编码

十进制数	8421 码	2421 码	余 3 码	Gray 码
0	0000	0000	0011	0000
1	0001	0001	0100	0001
2	0010	0010	0101	0011
3	0011	0011	0110	0010
4	0100	0100	0111	0110
5	0101	1011	1000	1110
6	0110	1100	1001	1010
7	0111	1101	1010	1011
8	1000	1110	1011	1001
9	1001	1111	1100	1000

1. 8421 码

8421 码又称为自然(Nature)BCD 码，简称 NBCD 码，4 位二进制代码的位权从高到低

分别为8、4、2、1，这种编码的主要特点如下：

① 它是一种有权码，设其各位的值为$b_3b_2b_1b_0$，则它所表示的十进制数$D=8b_3+4b_2+2b_1+1b_0$。

② 简单直观，每个代码与它所代表的十进制数之间符合二进制数和十进制数相互转换的规则。

③ 不允许出现1010～1111，这6个代码在8421码中是非法码。

注意：尽管在8421码中0～9十个数码的表示形式与用二进制表示的形式一样，但这是两个完全不同的概念，不能混淆。例如，一个两位的十进制数39，它可以表示为$(0011\ 1001)_{8421}$与100111B，这两者是完全不同的。

2. 2421码

2421码其编码各位的位权从高到低分别为2、4、2、1，它的主要特点如下：

① 它也是一种有权码，所表示的十进制数$D=2b_3+4b_2+2b_1+1b_0$。

② 它又是一种对9的自补码，即某数的2421码，只要自身按位取反，就能得到该数对9补数的2421码。例如，3的2421码是0011，3对9的补数是6，而6的2421码是1100，即将3的2421码自身按位取反可得到6的2421码。在十进制运算中，采用自补码，可以使运算器线路简化。

③ 不允许出现0101～1010，这6个代码在2421码中是非法码。

对于有权码来说，当规定各位的权不同时，可以有多种不同的编码方案，例如还有4221码、4421码、5421码和84-2-1码等多种编码方案。

3. 余3码

余3码是一种无权码，从表2-7中可以看出，余3码是在8421码的基础上加0011形成的，因每个数都余3，故称余3码，其主要特点如下：

① 它是一种无权码，在这种编码中各位的“1”不表示一个固定的十进制数值，因而不直观，且容易搞错。

② 它也是一种对9的自补码。

③ 不允许出现0000～0010和1101～1111，这6个代码在余3码中是非法码。

4. 格雷码(Gray码)

十进制Gray码的方案有很多种，表2-7中列出的只是其中的一种。Gray码可以避免在计数时发生中间错误，所以也被称为可靠性编码。其主要特点如下：

① 它也是一种无权码。

② 从一种代码变到相邻的下一种代码时，只有一个二进制位的状态在发生变化。

③ 具有循环特性，即首尾两个数的Gray码也只有一个二进制位不同，因此Gray码又称为循环码。

④ 十进制Gray码也有6个代码为非法码，视具体方案而定。

2.4.2 十进制数串

十进制数在计算机中是以数串的形式存储和处理的。十进制数串的长度是可变的，不

受定点数和浮点数统一格式的约束。十进制数在计算机内有两种表示形式：非压缩的十进制数和压缩的十进制数。

1. 非压缩的十进制数串

非压缩的十进制数串实际上就是前述的字符串，即一个字节存放一个十进制数或符号的 ASCII 码。在主存中，这样的十进制数串占用连续的多个字节。为了指明一个数串，需要给出该数串在主存中的起始地址和串长。

非压缩的十进制数串又根据符号所处的位置，分成前分隔式数字串和后嵌入式数字串两种格式。在前分隔式数字串中，符号位占用单独一个字节，放在数值位之前，正号对应的 ASCII 码为 2BH，负号对应的 ASCII 码为 2DH。在后嵌入式数字串中，符号位不单独占用一个字节，而是嵌入到最低一位数字里。若数串为正，则最低一位数字 0～9 的 ASCII 码不变(30H～39H)；若数串为负，把负号变为 40H，并将其与最低数值位相加，此时数字 0～9 的ASCII 码变为 70H～79H。

非压缩的十进制数串主要应用于非数值处理，而对十进制数的算术运算是很不方便的。因为每一字节中只有低 4 位表示数值，而高 4 位在算术运算时不具有数值的意义。

2. 压缩的十进制数串

压缩的十进制数串，一个字节可存放两位 BCD 码表示的十进制数，既节省了存储空间，又便于直接进行十进制算术运算，是广泛采用的表示方式。

在主存中，一个压缩的十进制数串占用连续的多个字节，每位数字仅占半个字节，其值常用 8421 码表示。符号位也占半个字节，并存放在最低数值位之后，通常用 CH 表示正号，DH 表示负号。在这种表示中，规定数字的个数加符号位之和必须为偶数；当和为奇数时，应在最高数值位之前补 0H(即第一个字节的高半字节为 0000)。

例 2-16　+123 表示为：

0001	0010	0011	1100

−2648 表示为：

0000	0010	0110	0100	1000	1101

要指明一个压缩的十进制数串，也必须给出它在主存中的首地址和串长。

2.5　不同类型的数据表示举例

前面已经讨论了许多种不同类型的数据表示，这里将以实际广泛应用的 C 语言中的数据类型和现代微型计算机中的数据表示作为实例进行介绍。

2.5.1　C 语言中的数据表示

C 语言的基本数据类型有整型数据、实型数据、字符型数据等，下面仅讨论整数类型和实数类型的表示。

1. 整数类型

C语言中支持多种整数类型,二进制整数分为无符号整数和带符号整数。

(1) 无符号整数

无符号整数对应 unsigned short、unsigned int、unsigned long 等类型。以 PC 为例,unsigned short(无符号短整型)和 unsigned int(无符号基本整型)字长为 16 位,数的范围是 0～65 535;unsigned long(无符号长整型)字长为 32 位,数的范围是 0～4 294 967 295。

(2) 带符号整数

带符号整数对应 short、int、long 等类型。以 PC 为例,short(短整型)和 int(基本整型)字长为 16 位,数的范围为 −32 768～32 767;long(长整型)字长为 32 位,数的范围为 −2 147 483 648～2 147 483 647。

2. 实数(浮点数)类型

C语言中有 float 和 double 两种不同浮点数类型,分别对应 IEEE 754 单精度浮点数格式(32 位)和双精度浮点数格式(64 位),前者数的范围是 $-3.4\times10^{38}\sim3.4\times10^{38}$,后者数的范围是 $-1.79\times10^{308}\sim1.79\times10^{308}$,相应的十进制有效数字分别为 6 位和 16 位。

浮点数均为带符号浮点数,没有无符号浮点数。

3. 不同数据类型之间的转换

数据类型的转换有以下 3 种基本形式:

① 同一类型但长度不同的数据间的转换。

② 定点方式与浮点方式间的转换。

③ 整型数中的带符号格式与无符号格式间的转换。

双目运算符两侧的操作数的类型必须一致,所得计算结果的类型与操作数的类型一致。如果一个运算符两边的操作数类型不同,则系统将自动按照转换规律先对操作数进行类型转换再进行运算,通常数据之间转换遵循的原则是"类型提升",即较低类型转换为较高类型。例如,float 型和 double 型数据参加运算,虽然它们同为实型,但两者的精度不同,需要先将 float 型转换成 double 型再进行运算,结果为 double 型。再如,一表达式中包含 int、long 和 double 类型的变量和数据,则表达式最后的运算结果是 double 型,3 种类型数据转换规律为 int→long→double。所有这些转换都是由系统自动进行的,这种转换通常称为隐式转换。

类型提升(升格)时,其值保持不变。例如,在将 16 位数与 32 位数相加之前,必须将 16 位数转换成 32 位数形式,这被称为"符号扩展",即用符号位来填充所有附加位。

当较高类型的数据转换成较低类型的数据时,称为降格,降格时就可能失去一部分信息。

除了隐式转换外,还有一种转换称为显式转换,这是一种强制转换类型机制。显式转换实际上是一种单目运算,其一般形式为:

(数据类型名)表达式

显式转换把后面的表达式运算结果的类型强制转换为要求的类型,而不管类型的高低。

要转换的表达式用括号括起来，例如，(int)$(x+y)$与(int)$x+y$是不同的，后者相当于(int)$(x)+y$，也就是说，只将x转换成整型，然后与y相加。

2.5.2 现代微型计算机系统中的数据表示

现代的微型计算机系统大多采用 Intel 系列的微处理器，近年来，Intel 的微处理器有了极大的发展，从 80386 到 80486、Pentium、Pentium MMX、Pentium Pro、Pentium Ⅱ、Pentium Ⅲ直至 Pentium 4，形成了 IA(Intel Architecture)-32 结构。而最新的 Itanium 处理器是目前 Intel 公司推出的最高端处理器，是第一款真正 64 位产品(IA-64)。下面以 IA-32 结构为例，介绍现代微机系统的数据表示。

IA-32 结构的基本数据类型是字节、字、双字(DWORD)、四字(QWORD)和双四字(DQWORD)，如图 2-9 所示。

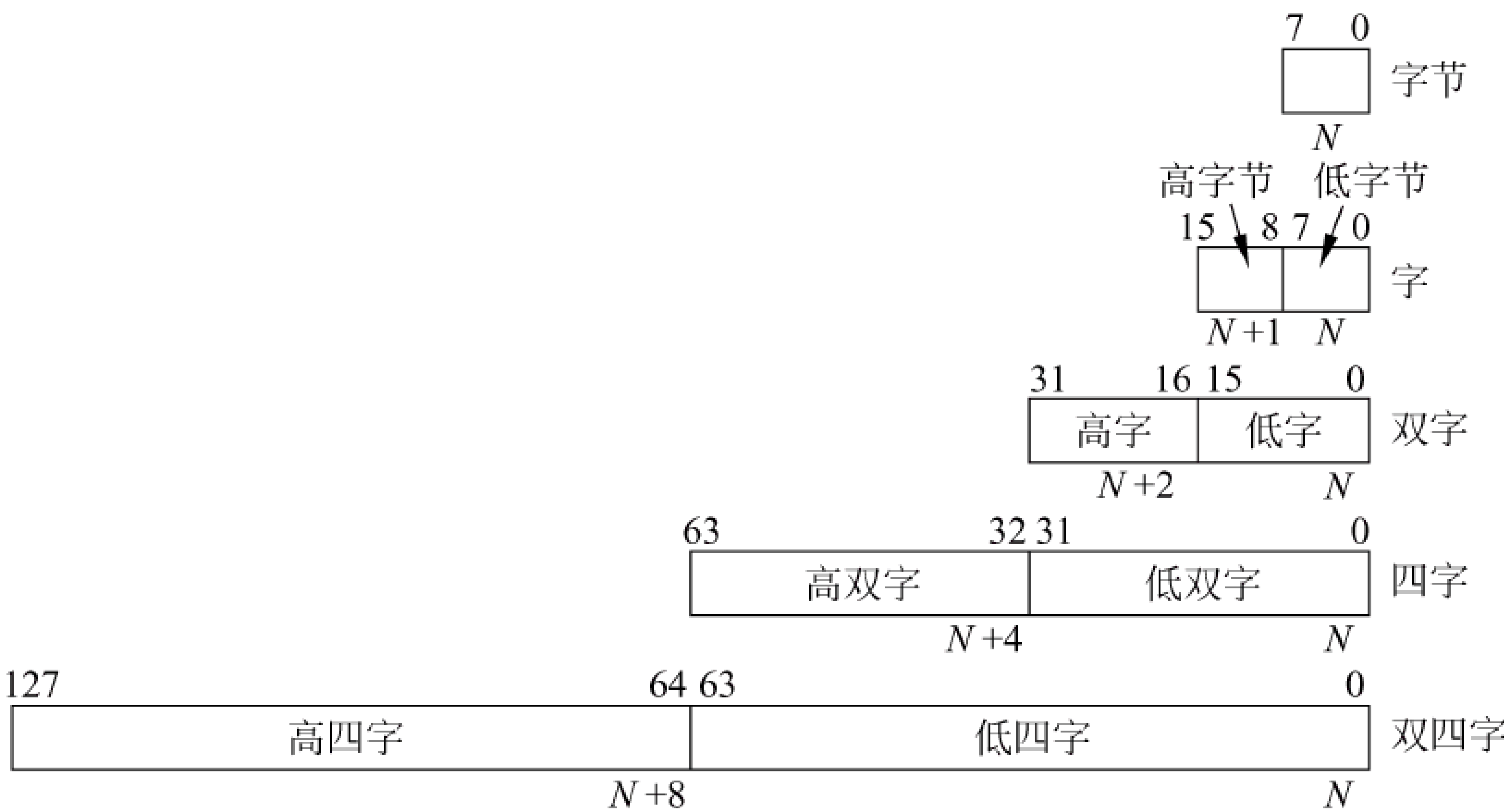


图 2-9 IA-32 结构的基本数据类型

注意：在 IA-32 结构的所有机器中，一个字都等于 16 位(两个字节)。双字是 4 字节(32 位)，四字是 8 字节(64 位)，双四字是 16 字节(128 位)。

四字是在微处理器 80486 中引入 IA-32 结构的，双四字是在具有 SSE 扩展的微处理器 Pentium Ⅲ中引入的。

1. 无符号整数

无符号整数是包含字节、字、双字和四字的无符号的二进制数。无符号整数的范围，对于字节，是 0～255；对于字，是 0～65 535；对于双字，是 $0\sim2^{32}-1$；对于四字，是 $0\sim2^{64}-1$。

2. 带符号整数

带符号整数是包含字节、字、双字和四字的带符号的二进制定点整数。所有带符号整数的数据类型都以补码形式表示，符号位是最高位(MSB)。正数的符号位为"0"，负数的符号位为"1"。带符号整数的范围，对于字节，是－128～＋127；对于字，是－32 768～＋32 767；对于双字，是 $-2^{31}\sim+2^{31}-1$；对于四字，是 $-2^{63}\sim+2^{63}-1$。

3. 浮点数

IA-32 结构定义和操作 3 种浮点数据类型：单精度浮点数(短浮点数)、双精度浮点数(长浮点数)和扩展精度浮点数(临时浮点数)。

这些数据类型的格式与 IEEE 754 标准所规定的格式直接对应，在此不再重复。

4. 指针数据

指针是主存单元的地址，IA-32 结构定义了两种类型的指针：近(Near)指针(32 位)和远(Far)指针(48 位)，如图 2-10 所示。近指针是段内的 32 位偏移量(也称为有效地址)，段内寻址时用。远指针是一个 48 位的逻辑地址，包含 16 位段选择符和 32 位的偏移量，用于跨段访问。



图 2-10　指针数据类型

5. 串数据

串是位、字节、字或双字的连续序列。串数据包括位串和字节串。位串是指连续的位，它能从任一字节的任一位置开始，位串长度可达 $2^{32}-1$ 位。字节串包含连续的字节、字或双字，长度为 0～$2^{32}-1$ 字节(4GB)。

6. BCD 数

IA-32 结构中所指的 BCD 码实际上是指 8421 码。BCD 数又分成未拼装的 BCD (UBCD)数和拼装的 BCD 数两种。UBCD 数的一个字节仅包含一位十进制数，在 3～0 位上；而经过拼装的 BCD 数，一个字节包含两位十进制数，其低位在 3～0 位上，高位在 7～4 位上。

2.6　数据校验码

数据在存取和传送的过程中可能会发生错误，产生错误的原因可能有很多种，例如，设备的临界工作状态、外界高频干扰、收发设备中的间歇性故障以及电源偶然的瞬变现象等。为了减少和避免错误，除了需要提高硬件本身的可靠性外，还要在数据编码上找出路。

数据校验码是指那些能够发现错误或者能够自动纠正错误的数据编码，又称为“检错纠错编码”。任何一种编码都由许多码字构成，任意两个码字之间最少变化的二进制位数称为数据校验码的码距。例如，用 4 位二进制表示 16 种状态，则有 16 个不同的码字，此时码距

为1,即两个码字之间最少仅有一个二进制位不同(如0000～0001)。这种编码没有检错能力,因为当某一个合法码字中有一位或几位出错,就变成另一个合法码字了。

具有检、纠错能力的数据校验码的实现原理是:在编码中,除去合法的码字外,再加进一些非法的码字,当某个合法码字出现错误时就变成非法码字。合理地安排非法码字的数量和编码规则,就能达到纠错的目的。例如,若用4位二进制表示8个状态,其中只有8个码字是合法码字,而另8个码字为非法码字,此时码距为2。对于码距≥2的数据校验码,开始具有检错的能力。码距越大,检、纠错能力就越强,而且检错能力总是大于或等于纠错能力。

2.6.1 奇偶校验码

1. 奇偶校验概念

奇偶校验码是一种最简单的数据校验码,它的码距等于2,可以检测出一位错误(或奇数位错误),但不能确定出错的位置,也不能检测出偶数位错误。事实上一位出错的概率比多位同时出错的概率要高得多,所以虽然奇偶校验码的检错能力很低,但还是一种应用最广泛的校验方法,常用于存储器读、写检查或ASCII字符传送过程中的检查。

奇偶校验实现方法是:由若干位有效信息(如一个字节),再加上一个二进制位(校验位)组成校验码,如图2-11所示。校验位的取值(0或1)将使整个校验码中1的个数为奇数或偶数,所以有如下两种可供选择的校验规律:

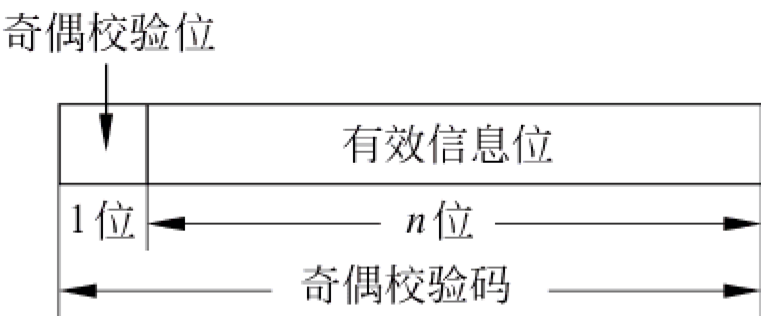


图2-11 奇偶校验码

① 奇校验——整个校验码(有效信息位和校验位)中1的个数为奇数。

② 偶校验——整个校验码中1的个数为偶数。

2. 简单奇偶校验

简单奇偶校验仅实现横向的奇偶校验,表2-8给出了几个字节的奇偶校验码的编码结果。

表2-8 奇偶校验码实例

有效信息(8位)	奇校验码(9位)	偶校验码(9位)
00000000	**1**00000000	**0**00000000
01010100	**0**01010100	**1**01010100
01111111	**0**01111111	**1**01111111
11111111	**1**11111111	**0**11111111

在表2-8所示的奇校验码或偶校验码中,最高一位为校验位,其余8位为信息位。在实际应用中,多采用奇校验,因为奇校验中不存在全0代码,在某些场合下更便于判别。

奇偶校验码的编码和校验是由专门的电路实现的，常见的有并行奇偶统计电路，如图2-12所示。这是一个由若干个异或门组成的塔形结构，同时给出了“偶形成”“奇形成”“偶校验出错”和“奇校验出错”等信号。

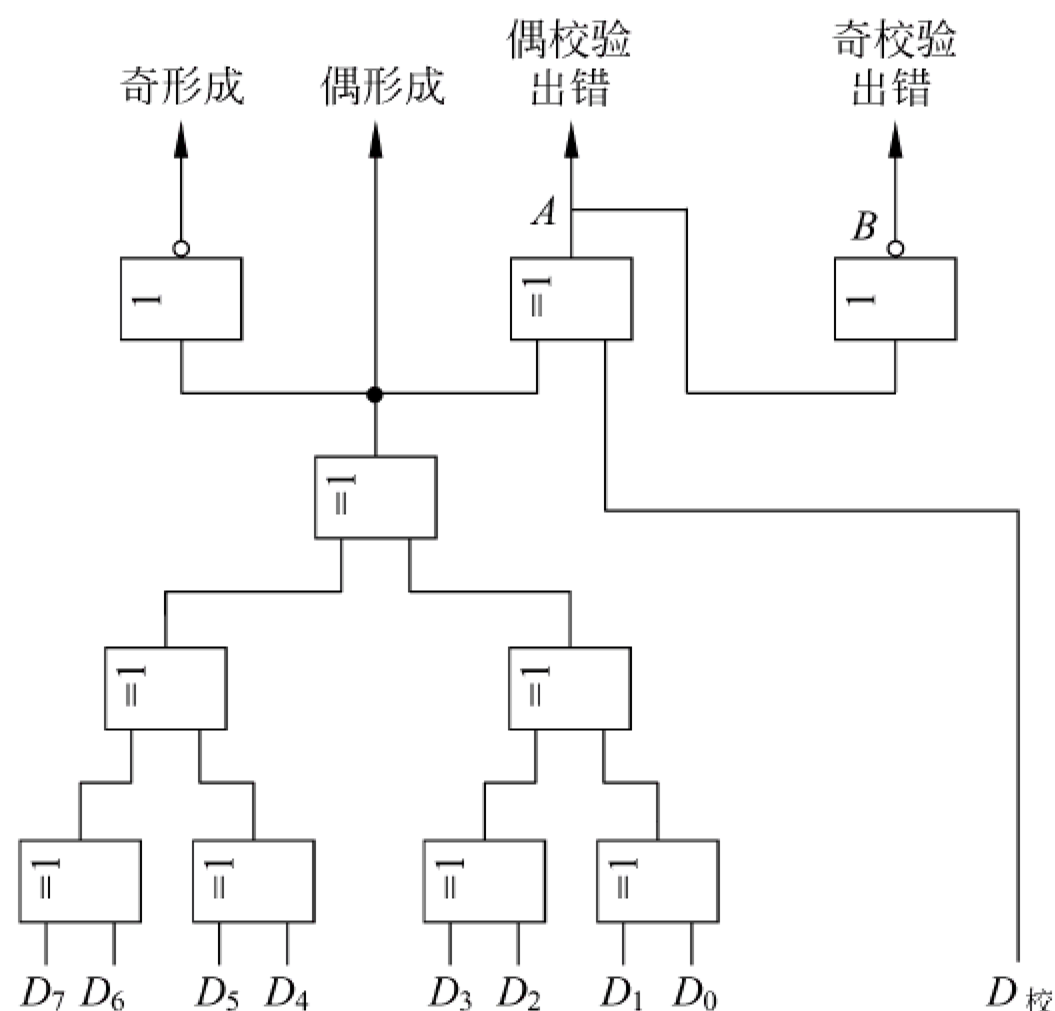


图2-12　奇偶校验位的形成及校验电路

$$偶形成=D_7\oplus D_6\oplus D_5\oplus D_4\oplus D_3\oplus D_2\oplus D_1\oplus D_0$$

$$奇形成=\overline{D_7\oplus D_6\oplus D_5\oplus D_4\oplus D_3\oplus D_2\oplus D_1\oplus D_0}$$

$$偶校验出错=D_校\oplus D_7\oplus D_6\oplus D_5\oplus D_4\oplus D_3\oplus D_2\oplus D_1\oplus D_0$$

$$奇校验出错=\overline{D_校\oplus D_7\oplus D_6\oplus D_5\oplus D_4\oplus D_3\oplus D_2\oplus D_1\oplus D_0}$$

下面以奇校验为例，说明对主存信息进行奇偶检验的全过程。

(1) 校验位形成

当要把一个字节的代码 $D_7 \sim D_0$ 写入主存时，就同时将它们送往奇偶校验逻辑电路，该电路产生的“奇形成”信号就是校验位。它将与8位代码一起作为奇校验码写入主存。

若 $D_7 \sim D_0$ 中有偶数个1，则“奇形成”=1；若 $D_7 \sim D_0$ 中有奇数个1，则“奇形成”=0。

(2) 校验检测

读出时，将读出的9位代码(8位信息位和1位校验位)同时送入奇偶校验电路检测。若读出代码无错，则“奇校验出错”=0；若读出代码中的某一位上出现错误，则“奇校验出错”=1，从而指示这个9位代码中一定有某一位出现了错误，但具体的错误位置是不能确定的。

3. 交叉奇偶校验

计算机在进行大量字节(数据块)传送时，不仅每一个字节有一个奇偶校验位做横向校验，而且全部字节的同一位也设置一个奇偶校验位做纵向校验，这种横向、纵向同时校验的方法称为交叉校验。

例如，4个字节组成一个信息块，纵、横向均约定为偶校验，各校验位取值如下：

	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0		横向校验位
第一字节	1	1	0	0	1	0	1	1	→	1
第二字节	0	1	0	1	1	1	0	0	→	0
第三字节	1	0	0	1	1	0	1	0	→	0
第四字节	1	0	0	1	0	1	0	1	→	0
	↓	↓	↓	↓	↓	↓	↓	↓		
纵向校验位	1	0	0	1	1	0	0	0		

交叉校验可以发现两位同时出错的情况,假设第二个字节的 A_6、A_4 两位均出错,第二个字节的横向校验位无法检出错误,但是第 A_6、A_4 位所在列的纵向校验位会显示出错,这与前述的简单奇偶校验相比要保险多了。

2.6.2 汉明校验码

汉明码是 Richard Hamming 于1950年提出的,目前仍是广泛采用的一种有效的校验码,主存的 ECC(Error Correcting Code)采用的就是与此类似的校验码。汉明码实际上是一种多重奇偶校验,其实现原理是:在有效信息位中加入几个校验位形成汉明码,使码距比较均匀地拉大,并把汉明码的每一个二进制位分配到几个奇偶校验组中。当某一位出错后,就会引起有关的几个校验位的值发生变化,这不但可以发现错误,还能指出错误的位置,为自动纠错提供了依据。

下面仅介绍能检测和自动校正一位错并能发现两位错的汉明码的编码原理。此时,校验位的位数 K 和信息位的位数 N 应满足:$2^{K-1} \geqslant N+K+1$。按该不等式,信息位 N 值和校验位 K 值的对应关系如表2-9所示。

表2-9 信息位 N 值与校验位 K 值的对应关系

N 值	最小 K 值	N 值	最小 K 值
1～3	4	26～56	7
4～10	5	57～119	8
11～25	6		

若汉明码的最高位号为 m,最低位号为1,即有 $H_m H_{m-1} \cdots H_2 H_1$,则此汉明码的编码规则通常为:

① 校验位和信息位之和为 m,每个校验位 P_i 在汉明码中被分到位号 2^{i-1} 的位置上,其余各位为信息位。

② 汉明码每一位 H_i 由多个校验位校验,其关系是被校验的每一位位号等于校验它的各校验位的位号之和,即汉明码的位号实质上是参与校验的各校验位权值之和。这样安排的目的,是希望校验的结果能正确地反映出错位的位号。

③ 在增大码距时,应使所有编码的码距尽量均匀地增大,以保证对所有代码的检测能力平衡的提高。

下面按以上原则介绍对一个字节信息进行汉明码编码和校验的过程。

(1) 编码

一个字节由8位二进制位组成,此时 $N=8$,从表2-9中查出 $K=5$,故汉明码的总位数

为13位，可表示为：

$$H_{13}H_{12}\cdots H_2H_1$$

5个校验位 $P_5\sim P_1$ 对应的汉明码位号应分别为 H_{13}、H_8、H_4、H_2、H_1，除 P_5 外，其余4位都满足 P_i 的位号等于 2^{i-1} 的关系，而 P_5 只能放在 H_{13} 上，因为它已经是汉明码的最高位了。因此，有如下排列关系：

$$P_5\quad D_8\quad D_7\quad D_6\quad D_5\quad P_4\quad D_4\quad D_3\quad D_2\quad P_3\quad D_1\quad P_2\quad P_1$$

校验位 $P_i(i=1\sim4)$ 的偶校验的结果为：

$$P_1=D_1\oplus D_2\oplus D_4\oplus D_5\oplus D_7$$
$$P_2=D_1\oplus D_3\oplus D_4\oplus D_6\oplus D_7$$
$$P_3=D_2\oplus D_3\oplus D_4\oplus D_8$$
$$P_4=D_5\oplus D_6\oplus D_7\oplus D_8$$

在上述4个公式中，不同信息位出现在 P_i 项中的次数是不一样的，其中 D_4 和 D_7 出现了3次，而 D_1、D_2、D_3、D_5、D_6、D_8 仅出现了两次，此时不同代码的汉明码的码距不等，为此再补充一位 P_5 校验位，使得：

$$P_5=D_1\oplus D_2\oplus D_3\oplus D_5\oplus D_6\oplus D_8$$

在这种安排下，每一位信息位都均匀地出现在3个 P_i 值的形成关系中。当任一信息位发生变化时，必将引起3个 P_i 值跟着变化，即合法汉明码的码距都为4。

(2) 校验

将接收到的汉明码按如下关系进行偶校验：

$$S_1=P_1\oplus D_1\oplus D_2\oplus D_4\oplus D_5\oplus D_7$$
$$S_2=P_2\oplus D_1\oplus D_3\oplus D_4\oplus D_6\oplus D_7$$
$$S_3=P_3\oplus D_2\oplus D_3\oplus D_4\oplus D_8$$
$$S_4=P_4\oplus D_5\oplus D_6\oplus D_7\oplus D_8$$
$$S_5=P_5\oplus D_1\oplus D_2\oplus D_3\oplus D_5\oplus D_6\oplus D_8$$

校验得到的结果值 $S_5\sim S_1$（指误字）能反映13位汉明码的出错情况：

① 当 $S_5\sim S_1$ 为00000时，表明无错。

② 当 $S_5\sim S_1$ 中仅有一位不为0时，表明是某一校验位出错或3位汉明码（包括信息位和校验位）同时出错。由于后一种出错的可能性很小，故认为是前一种错，出错位是该 S_i 对应的 P_i 位。

③ 当 $S_5\sim S_1$ 中有两位不为0时，表明是两位汉明码同时出错，此时只能发现错误，而无法确定出错的位置。

④ 当 $S_5\sim S_1$ 中有3位不为0时，表明是1位信息位出错或3位校验位同时出错，由于后一种错误的可能性很小，故认为是前一种错。出错位的位号由 $S_4\sim S_1$ 这4位代码值指明，此时不仅能检查出一位错，而且能准确的定位，因此可以纠正这个错误（将该位变反）。

⑤ 当 $S_5\sim S_1$ 中有4位或5位不为0时，表明出错情况严重，系统工作可能出现故障，应检查系统硬件的正确性。

②和④两种出错的情况列于表2-10中。若表中仅有一个 S_i 不为0，则表示 P_i 出错，因为是校验位出错，故此时并不需要校正它们。当5个 S_i 位有3个为1时，则表示是某一信息位 D_i 出错。出错信息位的汉明码位号由 $S_4\sim S_1$ 这4位的译码值指出（分别为12、11、10、9、

7、6、5、3)。例如,当 $S_5 \sim S_1 = 00111$ 时,$S_4 \sim S_1$ 的译码值为 7,即对应 H_7(也就是 D_4)位出错。

表 2-10 汉明码出错情况

汉明码	P_5	D_8	D_7	D_6	D_5	P_4	D_4	D_3	D_2	P_3	D_1	P_2	P_1
位号 / S 位	H_{13}	H_{12}	H_{11}	H_{10}	H_9	H_8	H_7	H_6	H_5	H_4	H_3	H_2	H_1
S_5	1	1	0	1	1	0	0	1	1	0	1	0	0
S_4	0	1	1	1	1	1	0	0	0	0	0	0	0
S_3	0	1	0	0	0	0	1	1	1	1	0	0	0
S_2	0	0	1	1	0	0	1	1	0	0	1	1	0
S_1	0	0	1	0	1	0	1	0	1	0	1	0	1

例 2-17 设有一个 8 位信息为 10101100,试求汉明编码的生成和校验过程。

(1) 编码生成

校验位长度为 5 位,按偶校验有:

$$P_1 = 0 \oplus 0 \oplus 1 \oplus 0 \oplus 0 = 1$$
$$P_2 = 0 \oplus 1 \oplus 1 \oplus 1 \oplus 0 = 1$$
$$P_3 = 0 \oplus 1 \oplus 1 \oplus 1 = 1$$
$$P_4 = 0 \oplus 1 \oplus 0 \oplus 1 = 0$$
$$P_5 = 0 \oplus 0 \oplus 1 \oplus 0 \oplus 1 \oplus 1 = 1$$

故可得到用二进制表示的汉明码为:

$$\underline{1}\ 1\ 0\ 1\ 0\ \underline{0}\ 1\ 1\ 0\ \underline{1}\ 0\ \underline{1}\ \underline{1}$$

下画线表示校验位在汉明码中的位置。

(2) 校验

上述汉明码经传送后若 $H_{11}(D_7)$ 位发生了错误,原码字就变为:

$$1\ 1\ \underset{\text{出错}}{\underline{1}}\ 1\ 0\ 0\ 1\ 1\ 0\ 1\ 0\ 1\ 1$$

检错的过程很简单,只要将接收到的码字重新进行偶校验即可:

$$S_1 = 1 \oplus 0 \oplus 0 \oplus 1 \oplus 0 \oplus 1 = 1$$
$$S_2 = 1 \oplus 0 \oplus 1 \oplus 1 \oplus 1 \oplus 1 = 1$$
$$S_3 = 1 \oplus 0 \oplus 1 \oplus 1 \oplus 1 = 0$$
$$S_4 = 0 \oplus 0 \oplus 1 \oplus 1 \oplus 1 = 1$$
$$S_5 = 1 \oplus 0 \oplus 0 \oplus 0 \oplus 1 \oplus 0 \oplus 1 \oplus 1 = 0$$

故指误字为 01011,其中低 4 位有效,相应的十进制数是 11,指出 H_{11} 出错。现在 H_{11} 错成了 1,纠错就是将 H_{11} 位取反让它恢复为 0,即:

$$\text{错误码:}\ 1\ 1\ \underline{1}\ 1\ 0\ 0\ 1\ 1\ 0\ 1\ 0\ 1\ 1$$
$$\downarrow$$
$$\text{纠正后:}\ 1\ 1\ \underline{0}\ 1\ 0\ 0\ 1\ 1\ 0\ 1\ 0\ 1\ 1$$

2.6.3 循环冗余校验码

除了奇偶校验码和汉明码外,在计算机网络、同步通信以及磁表面存储器中广泛使用循环冗余校验码(Cyclic Redundancy Check,CRC)。

循环冗余校验码是通过除法运算来建立有效信息位和校验位之间的约定关系的。假设,待编码的有效信息以多项式 $M(X)$ 表示,将它左移若干位后,用另一个约定的多项式 $G(X)$ 去除,所产生的余数 $R(X)$ 就是检验位。有效信息和检验位相拼接就构成了 CRC 码。当整个 CRC 码被接收后,仍用约定的多项式 $G(X)$ 去除,若余数为 0 表明该代码是正确的;若余数不为 0 则表明某一位出错,再进一步由余数值确定出错的位置,以便进行纠正。

1. 循环冗余校验码的编码方法

循环冗余校验码是由两部分组成的,如图 2-13 所示。左边为信息位,右边为校验位。若信息位为 N 位,校验位为 K 位,则该校验码被称为 $(N+K,N)$ 码。

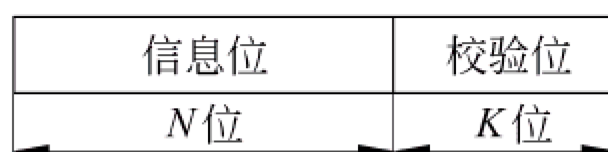


图 2-13 循环冗余校验码的格式

循环冗余校验码编码规律如下:

① 把待编码的 N 位有效信息表示为多项式 $M(X)$。

② 把 $M(X)$ 左移 K 位,得到 $M(X)\times X^K$,这样空出了 K 位,以便拼装 K 位余数(即校验位)。

③ 选取一个 $K+1$ 位的产生多项式 $G(X)$,对 $M(X)\times X^K$ 做模 2 除。

$$\frac{M(X)\times X^K}{G(X)}=Q(X)+\frac{R(X)}{G(X)}$$

④ 把左移 K 位以后的有效信息与余数 $R(X)$ 做模 2 加减,拼接为 CRC 码,此时的 CRC 码共有 $N+K$ 位。

$$M(X)\times X^K+R(X)=Q(X)\times G(X)$$

注意:CRC 校验技术中使用的模 2 运算是一种二进制运算,模 2 运算与四则运算不同之处在于它不用考虑进位和借位。

例 2-18 选择产生多项式为 1011,把 4 位有效信息 1100 编成 CRC 码。

$$M(X)=X^3+X^2=1100$$
$$M(X)\times X^3=X^6+X^5=1100000$$
$$G(X)=X^3+X+1=1011$$
$$\frac{M(X)\times X^3}{G(X)}=\frac{1100000}{1011}=1110+\frac{010}{1011}$$
$$M(X)\times X^3+R(X)=1100000+010=1100010$$

这种 CRC 码称为(7,4)码。

2. 循环冗余校验码的校验与纠错

把接收到的 CRC 码用约定的生成多项式 $G(X)$ 去除,如果正确,则余数为 0;如果某一位出错,则余数不为 0。不同的位数出错其余数不同,余数和出错位序号之间有唯一的对应关系。表 2-11 列出了(7,4)码的出错模式。

表 2-11　(7,4)码的出错模式($G(X)=1011$)

(7,4)码	A_1	A_2	A_3	A_4	A_5	A_6	A_7	余数	出错位
正确码	1	1	0	0	0	1	0	000	无
错误码	1	1	0	0	0	1	1	001	7
	1	1	0	0	0	0	0	010	6
	1	1	0	0	1	1	0	100	5
	1	1	0	1	0	1	0	011	4
	1	1	1	0	0	1	0	110	3
	1	0	0	0	0	1	0	111	2
	0	1	0	0	0	1	0	101	1

如果某一位出错,则余数不为0,对此余数补0后,当作被除数再继续除下去,余数将按表2-11的顺序循环。例如,第七位(A_7)出错,余数为001,把其补0后再除以$G(X)$,第二次余数为010,以后依次分别为100、011、110、111、101,然后又回到001,反复循环,这就是“循环码”词的来源。根据循环码的特征,一边对余数补0继续做模2除法,同时让被检测的校验码循环左移。当余数为101时,原来出错的A_7位已移到A_1的位置,通过异或门将其求反纠正,在下一次循环左移时送回A_7。所以,移满一个循环(7次),就得到一个纠正的码字。

3. 生成多项式的选择

生成多项式被用来生成CRC码,并不是任何一个$K+1$位多项式都可以作生成多项式用的,它应满足下列要求:

① 任何一位发生错误都应使余数不为0。

② 不同位发生错误应当使余数不同。

③ 对余数做模2除法,应使余数循环。

常用的生成多项式有多个,读者可从有关资料上查到可选生成多项式。在计算机和通信系统中广泛使用下述两个生成多项式,它们是:

$$G(X)=X^{16}+X^{15}+X^2+1$$
$$G(X)=X^{16}+X^{12}+X^6+1$$

习　题

2-1　设机器数的字长为8位(含1位符号位),分别写出下列各二进制数的原码、补码和反码。

0,　−0,　0.1000,　−0.1000,　0.1111,　−0.1111,　1101,　−1101

2-2　写出下列各数的原码、补码和反码。

$\frac{7}{16}$,　$\frac{4}{16}$,　$\frac{1}{16}$,　± 0,　$-\frac{1}{16}$,　$-\frac{4}{16}$,　$-\frac{7}{16}$

2-3　已知下列数的原码表示,分别写出它们的补码表示。

$$[X]_{原}=0.10100,\quad [X]_{原}=1.10111$$

2-4　已知下列数的补码表示,分别写出它们的真值。

$$[X]_{补}=0.10100,\quad [X]_{补}=1.10111$$

2-5　设一个二进制小数$X\geqslant 0$,表示成$X=0.A_1A_2A_3A_4A_5A_6$,其中$A_1\sim A_6$取1或0。

(1) 若要 $X>\frac{1}{2}$，则 $A_1\sim A_6$ 要满足什么条件？

(2) 若要 $X\geqslant\frac{1}{8}$，则 $A_1\sim A_6$ 要满足什么条件？

(3) 若要 $\frac{1}{4}\geqslant X>\frac{1}{16}$，则 $A_1\sim A_6$ 要满足什么条件？

2-6 设 $[X]_原=1.A_1A_2A_3A_4A_5A_6$。

(1) 若要 $X>-\frac{1}{2}$，则 $A_1\sim A_6$ 要满足什么条件？

(2) 若要 $-\frac{1}{8}\geqslant X\geqslant-\frac{1}{4}$，则 $A_1\sim A_6$ 要满足什么条件？

2-7 若将习题 2-6 中 $[X]_原$ 改为 $[X]_补$，那么结果如何？

2-8 一个 n 位字长的二进制定点整数，其中 1 位为符号位，分别写出在补码和反码两种情况下：

(1) 模数。 (2) 最大的正数。

(3) 最负的数。 (4) 符号位的权。

(5) −1 的表示形式。 (6) 0 的表示形式。

2-9 某计算机字长为 16 位，简述下列几种情况下所能表示数值的范围。

(1) 无符号整数。 (2) 用原码表示定点小数。

(3) 用补码表示定点小数。 (4) 用原码表示定点整数。

(5) 用补码表示定点整数。

2-10 某计算机字长为 32 位，试分别写出无符号整数和带符号整数（补码）的表示范围（用十进制数表示）。

2-11 假设机器数字长 8 位，若机器数为 81H，当它分别代表原码、补码、反码和移码时，等价的十进制整数分别是多少？

2-12 设计补码表示法的目的是什么？列表写出 +0、+25、+127、−127 及 −128 的 8 位二进制原码、反码、补码和移码表示，并将补码用十六进制表示出来。

2-13 十进制数 12345 用 32 位补码整数和 32 位浮点数（IEEE 754 标准）表示的结果各是什么（用十六进制表示）？

2-14 某浮点数字长为 12 位，其中，阶符为 1 位，阶码数值为 3 位；数符为 1 位，尾数数值为 7 位。阶码以 2 为底，阶码和尾数均用补码表示。它所能表示的最大正数是多少？最小规格化正数是多少？绝对值最大的负数是多少？

2-15 某浮点数字长为 16 位，其中，阶码部分为 6 位（含 1 位阶符），移码表示，以 2 为底；尾数部分为 10 位（含 1 位数符，位于尾数最高位），补码表示，规格化。分别写出下列情况的二进制代码与十进制真值。

(1) 非零最小正数。 (2) 最大正数。

(3) 绝对值最小负数。 (4) 绝对值最大负数。

2-16 一浮点数，其阶码部分为 p 位，尾数部分为 q 位，各包含 1 位符号位，均用补码表示；尾数基数 $r=2$，该浮点数格式所能表示数的上限、下限及非零的最小正数是多少？写出表达式。

2-17 若上题尾数基数 $r=16$，按上述要求写出表达式。

2-18 某浮点数字长为 32 位，格式如下。其中，阶码部分为 8 位，以 2 为底，移码表示；尾数部分一共 24 位（含 1 位数符），补码表示。现有一浮点代码为 $(8C5A3E00)_{16}$，试写出它所表示的十进制真值。

0　　　　7	8	9　　　　31
阶码	数符	尾数

2-19 试将 $(-0.1101)_2$ 用 IEEE 短浮点数格式表示出来。

2-20　将下列十进制数转换为 IEEE 短浮点数：

(1) 28.75　　(2) 624　　(3) －0.625

(4) ＋0.0　　(5) －1000.5

2-21　将下列 IEEE 短浮点数转换为十进制数：

(1) 11000000 11110000 00000000 00000000

(2) 00111111 00010000 00000000 00000000

(3) 01000011 10011001 00000000 00000000

(4) 01000000 00000000 00000000 00000000

(5) 01000001 00100000 00000000 00000000

(6) 00000000 00000000 00000000 00000000

2-22　对下列 ASCII 码进行译码。

1001001，　0100001，　1100001，　1110111，　1000101，　1010000，　1010111，　0100100

2-23　以下列形式表示$(5382)_{10}$。

(1) 8421 码。　(2) 余 3 码。　(3) 2421 码。　(4) 二进制数。

2-24　填写下列代码的奇偶校验位，现设为奇校验。

1 0 1 0 0 0 0 1

0 0 0 1 1 0 0 1

0 1 0 0 1 1 1 0

2-25　已知下面数据块约定：横向校验、纵向校验均为奇校验，指出至少有多少位出错。

	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0		校验位
	1	0	0	1	1	0	1	1	→	0
	0	0	1	1	0	1	0	1	→	1
	1	1	0	1	0	0	0	0	→	0
	1	1	1	0	0	0	0	0	→	0
	0	1	0	0	1	1	1	1	→	0
	↓	↓	↓	↓	↓	↓	↓	↓		
校验位	1	0	1	0	1	1	1	1		

2-26　求有效信息位为 01101110 的汉明校验码。

2-27　设计算机准备传送的信息是 1010110010001111，生成多项式是 X^5+X^2+1，计算校验位，写出 CRC 码。

第 3 章 指令系统

指令和指令系统是计算机中最基本的概念。指令是指示计算机执行某些操作的命令，一台计算机的所有指令的集合构成该机的指令系统，也称指令集。指令系统是计算机的主要属性，位于硬件和软件的交界面上。本章将讨论一般计算机的指令系统所涉及的基本问题。

3.1 指令格式

一台计算机指令格式的选择和确定涉及多方面的因素，如指令长度、地址码结构以及操作码结构等，是一个很复杂的问题，它与计算机系统结构、数据表示方法、指令功能设计等都密切相关。

3.1.1 机器指令的基本格式

一条指令就是机器语言的一个语句，它是一组有意义的二进制代码，指令的基本格式如下：

操作码字段	地址码字段

其中，操作码指明了指令的操作性质及功能，地址码则给出了操作数的地址。

指令的长度是指一条指令中所包含的二进制代码的位数，它取决于操作码字段的长度、操作数地址的个数及长度。指令长度与机器字长没有固定的关系，它可以等于机器字长，也可以大于或小于机器字长。在字长较短的小型、微型计算机中，大多数指令的长度可能大于机器的字长；而在字长较长的大型、中型计算机中，大多数指令的长度则往往小于或等于机器的字长。通常，把指令长度等于机器字长的指令称为单字长指令；指令长度等于半个机器字长的指令称为半字长指令；指令长度等于两个机器字长的指令称为双字长指令。

在一个指令系统中，若所有指令的长度都是相等的，就称该指令系统为定长指令字结构。定长结构指令系统控制简单，但不够灵活。若各种指令的长度随指令功能而异，就称该指令系统为变长指令字结构。现代计算机广泛采用变长指令字结构，指令的长度能短则短，需长则长，如 80x86 的指令长度从一个字节到十几个字节不等。变长结构指令系统灵活，能充分利用指令长度，但指令的控制较为复杂。

3.1.2 地址码结构

计算机执行一条指令所需要的全部信息都必须包含在指令中。对于一般的双操作数运算类指令来说，除去操作码(Operation Code)之外，指令还应包含以下信息：

① 第一操作数地址，用 A_1 表示。

② 第二操作数地址，用 A_2 表示。

③ 操作结果存放地址，用 A_3 表示。

④ 下一条将要执行指令的地址，用 A_4 表示。

这些信息可以在指令中明显地给出，称为显地址；也可以依照某种事先的约定，用隐含的方式给出，称为隐地址。下面从地址结构的角度来介绍几种指令格式。

1. 四地址指令

前述的4个地址信息都在地址字段中明显地给出，其指令的格式为：

OP	A_1	A_2	A_3	A_4

指令的含义：

$$(A_1)\text{OP}(A_2)\rightarrow A_3$$

$$A_4 = \text{下一条将要执行指令的地址}$$

其中，OP表示具体的操作，A_i表示地址，(A_i)表示存放于该地址中的内容。

这种格式的主要优点是直观，下一条指令的地址明显。但是，最严重的缺点是指令的长度太长，如果每个地址为16位，整个地址码字段就要长达64位，所以这种格式是不切实际的。

2. 三地址指令

正常情况下，大多数指令按顺序依次被从主存中取出来执行，只有在遇到转移指令时，程序的执行顺序才会改变。因此，可以用一个程序计数器(Program Counter，PC)来存放指令地址。通常每执行一条指令，PC就自动加1(设每条指令只占一个主存单元)，直接得到将要执行的下一条指令的地址。这样，指令中就不必再明显地给出下一条指令的地址了。三地址指令格式为：

OP	A_1	A_2	A_3

指令的含义：

$$(A_1)\text{OP}(A_2)\rightarrow A_3$$

$$(\text{PC})+1\rightarrow \text{PC}(\text{隐含})$$

执行一条三地址的双操作数运算指令，至少需要访问4次主存。第一次取指令本身，第二次取第一操作数，第三次取第二操作数，第四次保存运算结果。

这种格式省去了一个地址，但指令长度仍比较长，所以只在字长较长的大型、中型计算机中使用，小型、微型计算机中很少使用。

3. 二地址指令

三地址指令执行完后，主存中的两个操作数均不会被破坏。然而，通常并不一定需要完整的保留两个操作数。例如，可让第一操作数地址同时兼作存放结果的地址（目的地址），这样即得到了二地址指令，其格式为：

OP	A_1	A_2

指令的含义：

$$(A_1)OP(A_2) \rightarrow A_1$$

$$(PC)+1 \rightarrow PC(\text{隐含})$$

其中，A_1为目的操作数地址，A_2为源操作数地址。

注意：指令执行之后，目的操作数地址中原存的内容已被破坏了。

执行一条二地址的双操作数运算指令，同样至少需要访问4次主存。

4. 一地址指令

一地址指令顾名思义只有一个显地址，它的指令格式为：

OP	A_1

一地址指令只有一个地址，那么另一个操作数来自何方呢？指令中虽未明显给出，但按事先约定，这个隐含的操作数就放在一个专门的寄存器中。因为这个寄存器在连续运算时，保存着多条指令连续操作的累计结果，故称为累加寄存器（Accumulator，Acc）。

指令的含义：

$$(Acc)OP(A_1) \rightarrow Acc$$

$$(PC)+1 \rightarrow PC(\text{隐含})$$

执行一条一地址的双操作数运算指令，只需要访问两次主存。第一次取指令本身，第二次取第二操作数。第一操作数和运算结果都放在累加寄存器中，所以读取和存入都不需要访问主存。

5. 零地址指令

零地址指令格式中只有操作码字段，没有地址码字段，其格式为：

OP

零地址的算术逻辑类指令是用在堆栈计算机中的，堆栈计算机没有一般计算机中必备的通用寄存器，因此堆栈就成为提供操作数和保存运算结果的唯一场所。通常，参加算术逻辑运算的两个操作数隐含地从堆栈顶部弹出，送到运算器中进行运算，运算的结果再隐含地压入堆栈。有关堆栈的概念将在稍后介绍。

指令中地址个数的选取要考虑诸多的因素。从缩短程序长度、用户使用方便、增加操作并行度等方面来看，选用三地址指令格式较好；从缩短指令长度，减少访存次数、简化硬件设

计等方面来看,一地址指令格式较好。对于同一个问题,用三地址指令编写的程序最短,但指令长度(程序存储量)最长;而用二、一、零地址指令来编写程序,程序的长度一个比一个长,但指令的长度一个比一个短。表3-1给出了不同地址数指令的特点及适用场合。

表3-1 不同地址数指令的特点及适用场合

地址数量	程序长度	程序存储量	执行速度	适用场合
三地址	短	最大	一般	以向量、矩阵运算为主
二地址	一般	很大	很低	一般不宜采用
一地址	较长	较大	较快	连续运算,硬件结构简单
零地址	最长	最小	最低	嵌套、递归问题

前面介绍的操作数地址都是指主存单元的地址,实际上许多操作数可能是存放在通用寄存器里的。计算机在CPU中设置了相当数量的通用寄存器,用它们来暂存运算数据或中间结果,这样可以大大减少访存次数,提高计算机的处理速度。实际使用的二地址指令多为二地址R(通用寄存器)型,一般通用寄存器数量有8~32个,其地址(或称寄存器编号)有3~5位就可以了。由于二地址R型指令的地址码字段很短,且操作数就在寄存器中,所以这类指令的程序存储量最小,程序执行速度最快,在小型、微型计算机中被大量使用。

3.1.3 指令的操作码

指令系统中的每一条指令都有一个唯一确定的操作码,指令不同,其操作码的编码也不同。通常,希望用尽可能短的操作码字段来表达全部的指令。指令操作码的编码可以分为规整型和非规整型两类编码。

1. 规整型编码(定长编码)

这是一种最简单的编码方法,操作码字段的位数和位置是固定的。为了能表示整个指令系统中的全部指令,指令的操作码字段应当具有足够的位数。

假定指令系统共有 m 条指令,指令中操作码字段的位数为 N 位,则有如下关系式:

$$m \leqslant 2^N$$

所以,

$$N \geqslant \log_2 m$$

定长编码对于简化硬件设计、减少指令译码的时间是非常有利的,在字长较长的大型、中型计算机及超级小型计算机上广泛采用。例如,IBM 370机(字长32位)中采用的就是这种方式。IBM 370机的指令可分为3种不同的长度形式:半字长指令(16位)、单字长指令(32位)和一个半字长指令(48位),共有5种格式,如图3-1所示。

从图3-1可以看出,在IBM 370机中不论指令的长度为多少位,其操作码字段一律都是8位。8位操作码允许容纳256条指令。而实际上在IBM 370机中仅有183条指令,存在着极大的信息冗余,这种信息冗余的编码也称为非法操作码。

2. 非规整型编码(变长编码)

变长编码的操作码字段的位数不固定,且分散地放在指令字的不同位置上。这种方式能

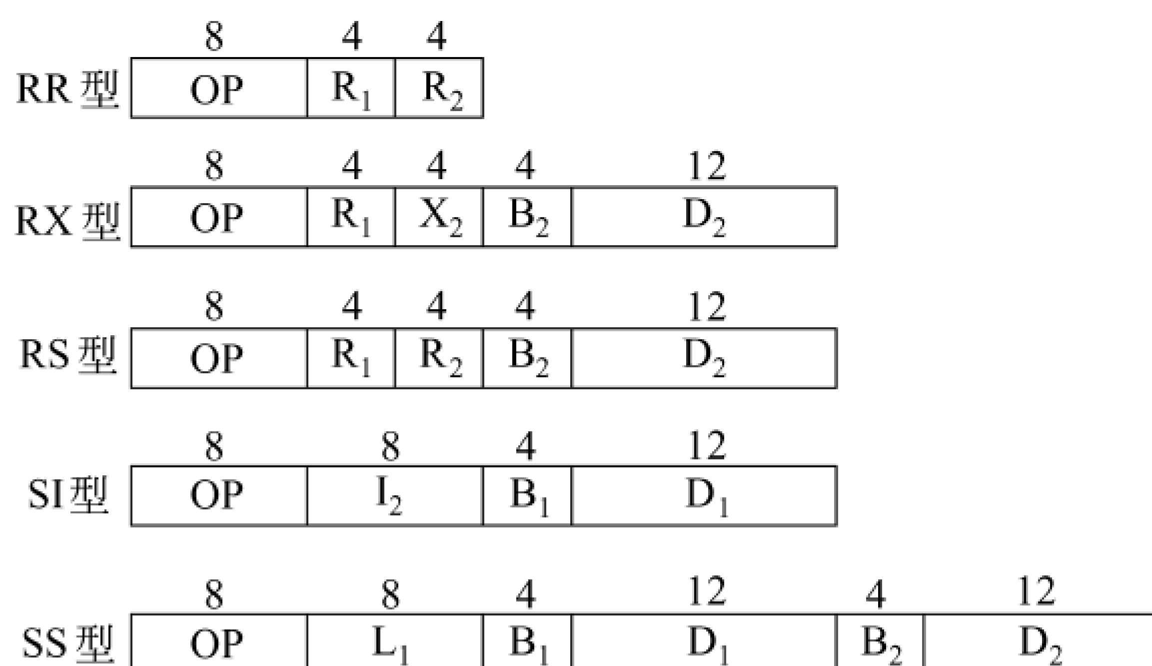


图 3-1　IBM 370 机的指令格式

够有效地压缩指令中操作码字段的平均长度，在字长较短的小型、微型计算机上广泛采用。例如，PDP-11 机(字长 16 位)中采用的就是这种方式。PDP-11 机的指令分为单字长、二字长、三字长 3 种，操作码字段占 4～16 位不等，可遍及整个指令长度，其指令格式如图 3-2 所示。

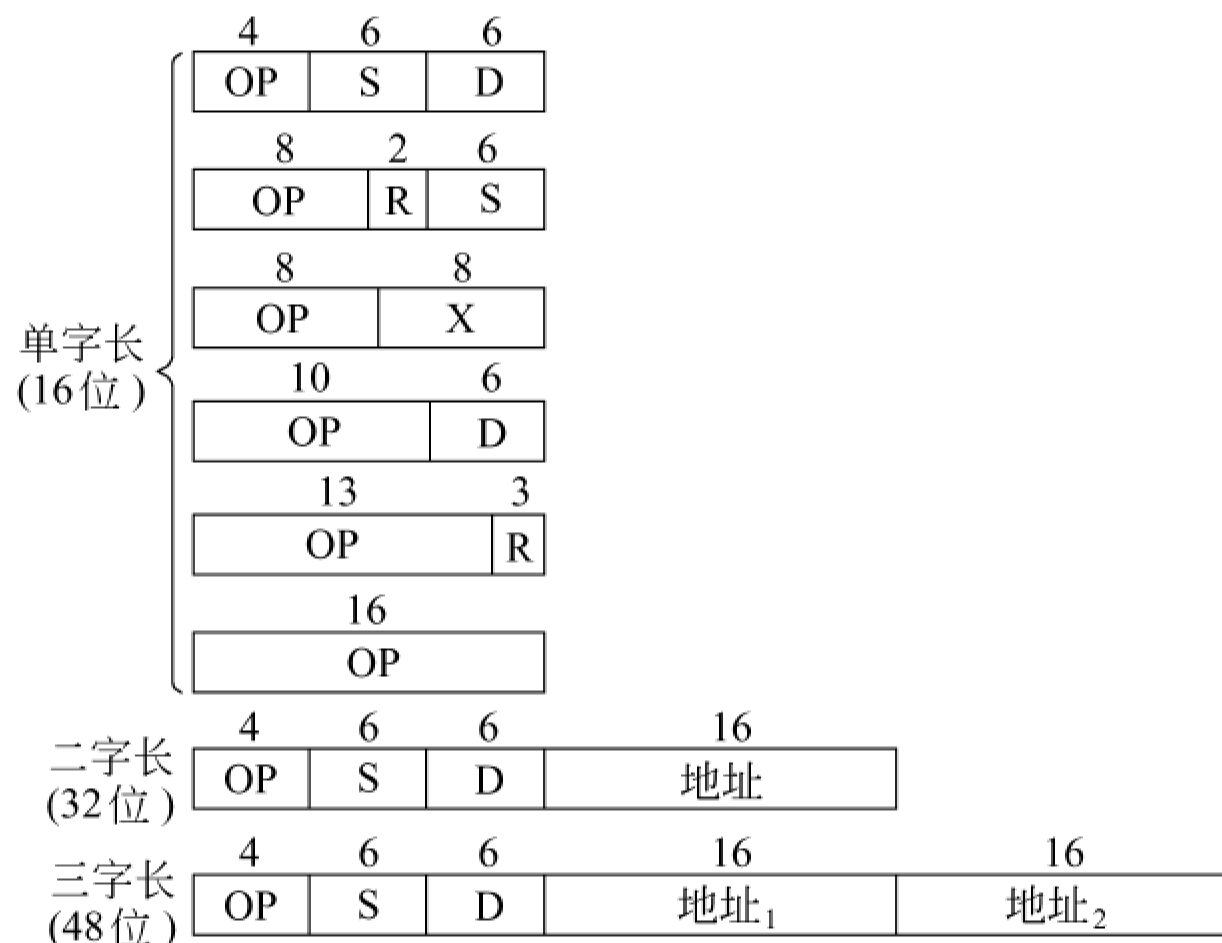
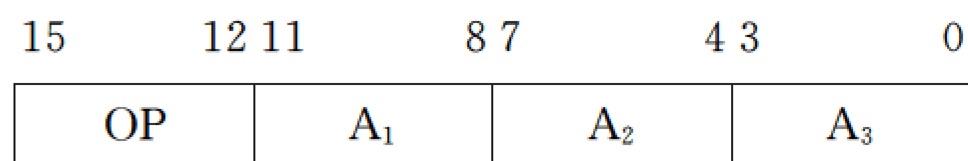


图 3-2　PDP-11 机的指令格式

显然，操作码字段的位数和位置不固定将增加指令译码和分析的难度，使得控制器的设计复杂化。

最常用的非规整型编码方式是扩展操作码法。因为如果指令长度一定，则地址码与操作码字段的长度是相互制约的。为了解决这一矛盾，让操作数地址个数多的指令(三地址指令)的操作码字段短些，操作数地址个数少的指令(一或零地址指令)的操作码字段长些，这样既能充分地利用指令的各个字段，又能在不增加指令长度的情况下扩展操作码的位数，使它能表示更多的指令。例如，设某计算机的指令长度为 16 位，操作码字段为 4 位，有 3 个 4 位的地址码字段，其格式为：



如果按照定长编码的方法，4 位操作码最多只能表示 16 条不同的三地址指令。假设指

令系统中不仅有三地址指令,还有二地址指令、一地址指令和零地址指令,利用扩展操作码法可以使在指令长度不变的情况下,指令的总数远远大于16条。例如,指令系统中要求有15条三地址指令、15条二地址指令、15条一地址指令和16条零地址指令,共61条指令。显然,只有4位操作码是不够的,解决的方法就是向地址码字段扩展操作码的位数。扩展的方法如下:

① 4位操作码的编码0000～1110定义了15条三地址指令,留下1111作为扩展窗口,与下一个4位(A_1)组成一个8位的操作码字段。

② 8位操作码的编码11110000～11111110定义了15条二地址指令,留下11111111作为扩展窗口,与下一个4位(A_2)组成一个12位的操作码字段。

③ 12位操作码的编码111111110000～111111111110定义了15条一地址指令,扩展窗口为111111111111,与A_3组成16位的操作码字段。

④ 最后,16条零地址指令由16位操作码的编码1111111111110000～1111111111111111给出。

根据指令系统的要求,扩展操作码的组合方案可以有很多种,但有以下两点要注意:

- 不允许短码是长码的前缀,即短码不能与长码的开始部分的代码相同,否则将无法保证解码的唯一性和实时性。
- 各条指令的操作码一定不能重复,而且各类指令的格式安排应统一规整。

3.2 寻址技术

所谓寻址,指的是寻找操作数的地址或下一条将要执行的指令地址,寻址技术是计算机设计中硬件对软件最早提供支持的技术之一。寻址技术包括编址方式和寻址方式。

3.2.1 编址方式

在计算机中,编址方式是指对各种存储设备进行编码的方式。

1. 编址

通常,指令中的地址码字段将指出操作数的来源和去向,而操作数则存放在相应的存储设备中。在计算机中需要编址的设备主要有CPU中的通用寄存器、主存储器和输入输出设备3种。

要对寄存器、主存储器和输入输出设备进行访问,首先必须对它们进行编址。就像一个大楼有许多房间,首先必须给每一个房间编上一个唯一的号码,人们才能据此找到需要的房间一样。

如果存储设备是CPU中的通用寄存器,那么在指令字中应给出寄存器编号;如果是主存的一个存储单元,那么在指令字中应给出该主存单元的地址;如果是输入输出设备(接口)中的一个寄存器,那么指令字中应给出设备编号或设备端口地址或设备映像地址(与主存地址统一编址时)。

2. 编址单位

目前常用的编址单位有字编址、字节编址和位编址。

(1) 字编址

字编址是实现起来最容易的一种编址方式,这是因为每个编址单位与访问单位相一致,即每个编址单位所包含的信息量(二进制位数)与访问一次寄存器、主存所获得的信息量相同。早期的大多数机器都采用这种编址方式。

在采用字编址的机器中,每执行一条指令,程序计数器加1;每从主存中读出一个数据,地址计数器加1。这种控制方式实现起来简单,地址信息没有任何浪费。但它的主要缺点是不支持非数值应用,而目前在计算机的实际应用领域中,非数值应用已超过数值应用。

(2) 字节编址

目前使用最普遍的编址方式是字节编址,这是为了适应非数值应用的需要。字节编址方式使编址单位与信息的基本单位(一个字节)相一致,这是它的最大优点。然而,如果主存的访问单位也是一个字节的话,那么主存的带宽就太窄了,所以编址单位和主存的访问单位是不相同的。通常主存的访问单位是编址单位的若干倍。

在采用字节编址的机器中,如果指令长度是32位,那么每执行完一条指令,程序计数器要加4。如果数据字长是32位,当连续访问存储器时,每读写完一个数据字,地址寄存器要加4。由此可见,字节编址方式存在着地址信息的浪费。

(3) 位编址

有部分计算机系统采用位编址方式,如STAR-100巨型计算机等。这种编址方式的地址信息浪费更大。

3. 指令中地址码的位数

指令格式中每个地址码的位数是与主存容量和最小寻址单位(即编址单位)有关联的。主存容量越大,所需的地址码位数就越长。对于相同容量来说,如果以字节为最小寻址单位,那么地址码的位数就需要长些,但是可以方便地对每一个字符进行处理;如果以字为最小寻址单位(假定字长为16位或更长),那么地址码的位数可以减少,但对字符操作比较困难。例如,设某计算机主存容量为2^{20}个字节,机器字长32位,若最小寻址单位为字节(按字节编址),其地址码应为20位;若最小寻址单位为字(按字编址),其地址码只需18位。从减少指令长度的角度看,最小寻址单位越大越好;而从对字符或位的操作是否方便的角度看,最小寻址单位越小越好。

3.2.2 指令寻址和数据寻址

寻址可以分为指令寻址和数据寻址。寻找下一条将要执行的指令地址称为指令寻址,寻找操作数的地址称为数据寻址。指令寻址比较简单,它又可以细分为顺序寻址和跳跃寻址。而数据寻址方式种类较多,其最终目的都是寻找所需要的操作数。

顺序寻址可通过程序计数器加1,自动形成下一条指令的地址;跳跃寻址则需要通过程序转移类指令实现。

跳跃寻址的转移地址形成方式有3种:直接(绝对)、相对和间接寻址,它与下面介绍的数据寻址方式中的直接、相对和间接寻址是相同的,只不过寻找到的不是操作数的有效地址而是转移的有效地址。

3.2.3 基本的数据寻址方式

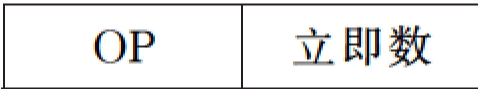
数据寻址方式是根据指令中给出的地址码字段寻找真实操作数地址的方式。一般情况下,由于指令长度的限制,指令中的地址码不会很长,而主存的容量却可能越来越大。以 IBM PC/XT 机为例,主存容量可达 1MB,而指令中的地址码字段最长仅 16 位,仅能直接访问主存的一小部分,而无法访问整个主存空间。就是在字长很长的大型机中,即使指令中能够拿出足够的位数来作为访问整个主存空间的地址,为了灵活方便地编制程序,也需要对地址进行必要的变换。指令中地址码字段给出的地址称为形式地址(用字母 A 表示),这个地址有可能不能直接用来访问主存。形式地址经过某种运算而得到的能够直接访问主存的地址称为有效地址(用字母 EA 表示)。从形式地址生成有效地址的各种方式称为寻址方式,即:

$$\text{指令中的形式地址} \xrightarrow{\text{寻址方式}} \text{有效地址}$$

每种计算机的指令系统都有自己的一套数据寻址方式,不同计算机的寻址方式的名称和含义并不统一,下面介绍大多数计算机常用的几种基本寻址方式。

1. 立即寻址

立即寻址是一种特殊的寻址方式,指令中在操作码字段后面的部分不是通常意义上的操作数地址,而是操作数本身。也就是说,数据就包含在指令中,只要取出指令,也就取出了可以立即使用的操作数,这样的数称为立即数,其指令格式为:



这种方式的特点是:在取指令时,操作码和操作数被同时取出,不必再次访问主存,从而提高了指令的执行速度。但是,因为操作数是指令的一部分,不能被修改,而且立即数的大小受到指令长度的限制,所以这种寻址方式灵活性最差,通常用于给某一寄存器或主存单元赋初值或提供一个常数。

2. 寄存器寻址

寄存器寻址指令的地址码部分给出某一个通用寄存器的编号 R_i,这个指定的寄存器中存放着操作数。寄存器寻址过程如图 3-3 所示,图中的 IR 表示指令寄存器,它的内容是从主存中取出的指令。操作数 S 与寄存器 R_i 的关系为:

$$S=(R_i)$$

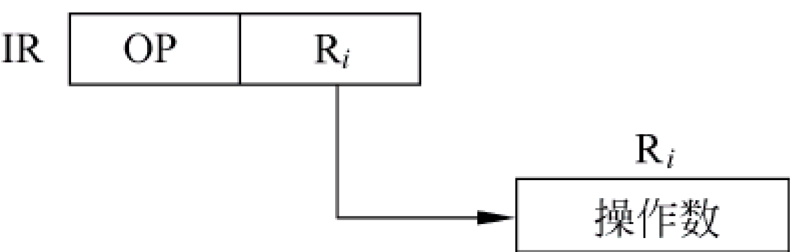


图 3-3 寄存器寻址过程

这种寻址方式具有两个明显的优点:

① 从寄存器中存取数据比从主存中存取快得多。

② 由于寄存器的数量较少,其地址码字段比主存单元地址字段短得多。

这种方式可以缩短指令长度,提高指令的执行速度,几乎所有的计算机都使用了寄存器寻址方式。

3. 直接寻址

指令中地址码字段给出的地址 A 就是操作数的有效地址，即形式地址等于有效地址：EA＝A。由于这样给出的操作数地址是不能修改的，与程序本身所在的位置无关，所以又称为绝对寻址方式。图 3-4 所示为直接寻址的示意图。操作数 S 与地址码 A 的关系为：

$$S=(A)$$

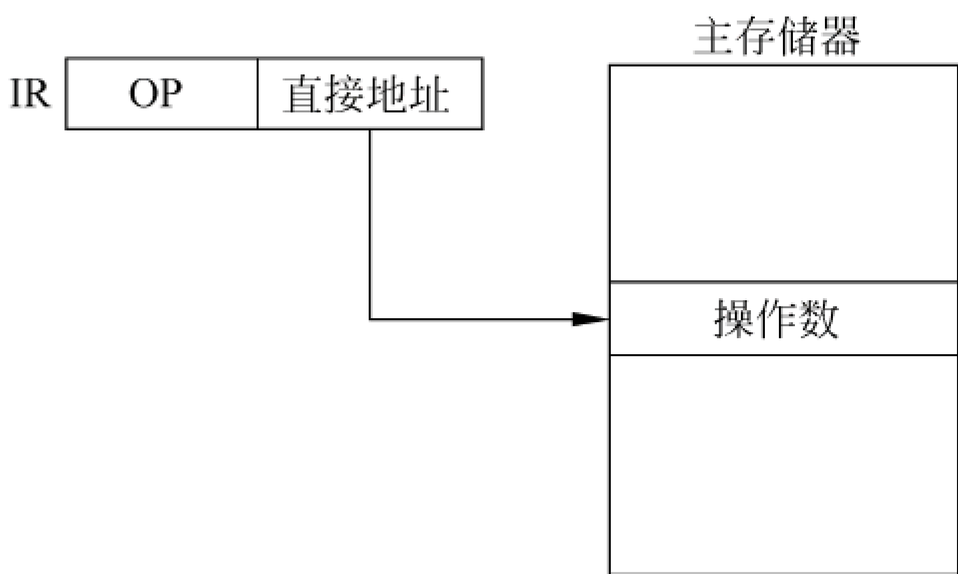


图 3-4　直接寻址过程

这种寻址方式不需做任何寻址运算，简单直观，也便于硬件实现，但地址空间受到指令中地址码字段位数的限制。

4. 间接寻址

间接寻址意味着指令中给出的地址 A 不是操作数的地址，而是存放操作数地址的主存单元的地址，简称操作数地址的地址。通常在指令格式中划出一位作为直接或间接寻址的标志位，间接寻址时标志位@＝1。

间接寻址中又有一级间接寻址和多级间接寻址之分。在一级间接寻址中，首先按指令的地址码字段先从主存中取出操作数的有效地址，即 EA＝(A)，然后再按此有效地址从主存中读出操作数，如图 3-5(a)所示。操作数 S 与地址码 A 的关系为：

$$S=((A))$$

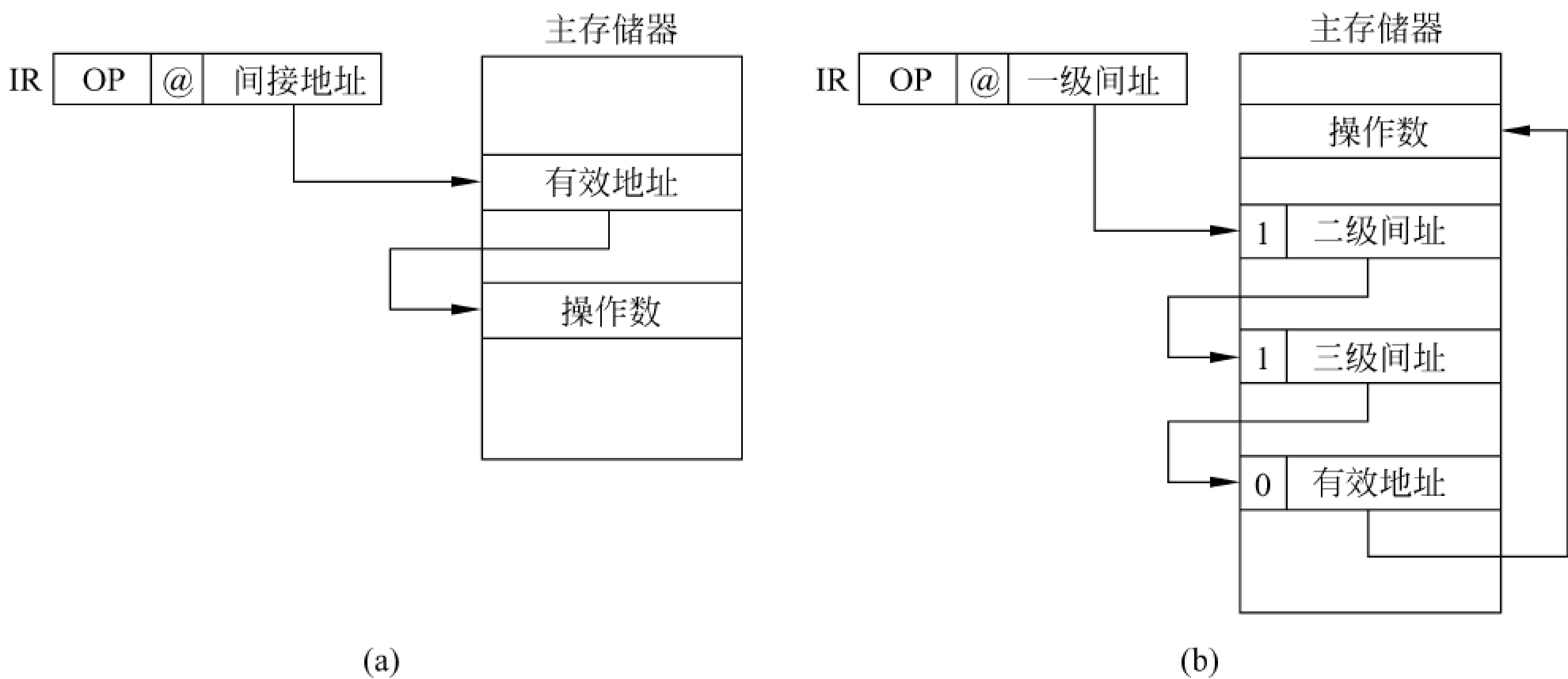


图 3-5　间接寻址过程

多级间接寻址为取得操作数需要多次访问主存,即使在找到操作数有效地址后,还需再访问一次主存才可得到真正的操作数,如图 3-5(b)所示。对于多级间接寻址来说,在寻址过程中所访问到的每个主存单元的内容中都应设有一个间址标志位。通常将这个标志放在主存单元的最高位。当该位为 1 时,表示这一主存单元中仍然是间接地址,需要继续间接寻址;当该位为 0 时,表示已经找到了有效地址,根据这个地址可以读出真正的操作数。

间接寻址要比直接寻址灵活得多,它的主要优点如下:

① 扩大了寻址范围,可用指令中的短地址访问大的主存空间。

② 可将主存单元作为程序的地址指针,用以指示操作数在主存中的位置。当操作数的地址需要改变时,不必修改指令,只须修改存放有效地址的那个主存单元的内容即可。

但是,间接寻址在取指之后至少需要两次访问主存才能取出操作数,降低了取操作数的速度。尤其是在多级间接寻址时,寻找操作数要花费相当多的时间,甚至可能发生间址循环。

5. 寄存器间接寻址

为了克服间接寻址中访存次数多的缺点,可采用寄存器间接寻址,即指令中的地址码给出某一通用寄存器的编号,在被指定的寄存器中存放操作数的有效地址,而操作数则存放在主存单元中,其寻址过程如图 3-6 所示。操作数 S 与寄存器号 R_i 的关系为:

$$S=((R_i))$$

这种寻址方式的指令较短,并且在取指后只须一次访存便可得到操作数,因此指令执行速度较间接寻址方式快,是一种使用广泛的寻址方式。

6. 变址寻址

变址寻址就是把变址寄存器 R_x 的内容与指令中给出的形式地址 A 相加,形成操作数有效地址,即 $EA=(R_x)+A$。R_x 的内容称为变址值,其寻址过程如图 3-7 所示。操作数 S 与地址码和变址寄存器的关系为:

$$S=((R_x)+A)$$

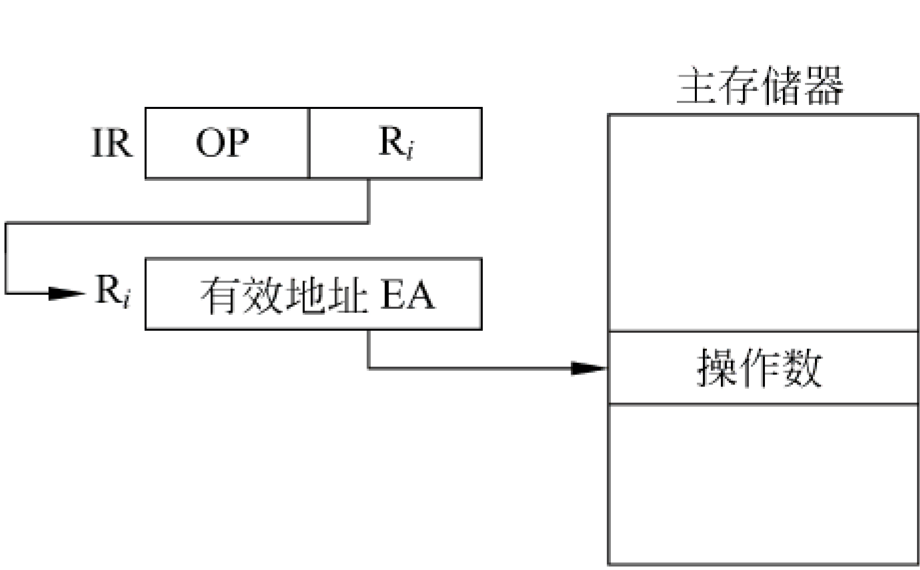


图 3-6 寄存器间接寻址过程

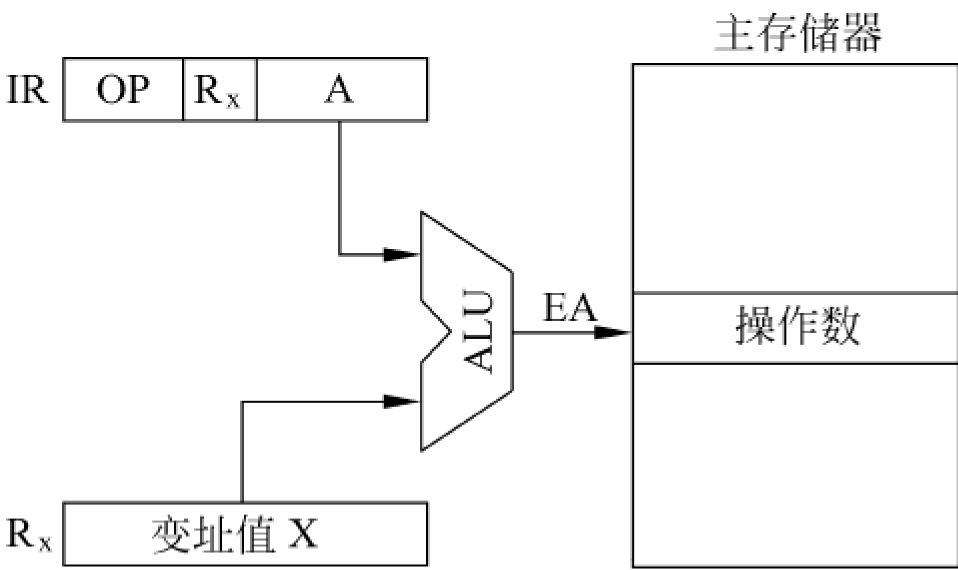


图 3-7 变址寻址过程

变址寻址是一种广泛采用的寻址方式,最典型的用法是将指令中的形式地址作为基准地址,而变址寄存器的内容作为修改量。在遇到需要频繁修改地址时,无须修改指令,只要修改变址值就可以了,这对于数组运算、字符串操作等成批数据处理是很有用的。例如,要

把一组连续存放在主存单元中的数据(首地址是A)依次传送到另一存储区(首地址为B)中,则只须在指令中指明两个存储区的首地址A和B(形式地址),用同一变址寄存器提供修改量K,即可实现(A+K)→B+K。变址寄存器的内容在每次传送之后自动地修改。

在具有变址寻址的指令中,除去操作码和形式地址外,还应具有变址寻址标志,当有多个变址寄存器时,还必须指明具体寻找哪一个变址寄存器。

7. 基址寻址

基址寻址是将基址寄存器 R_b 的内容与指令中给出的位移量D相加,形成操作数有效地址,即 $EA=(R_b)+D$。基址寄存器的内容称为基址值。指令的地址码字段是一个位移量,位移量可正、可负,如图3-8所示。操作数S与基址寄存器和地址码的关系为:

$$S=((R_b)+D)$$

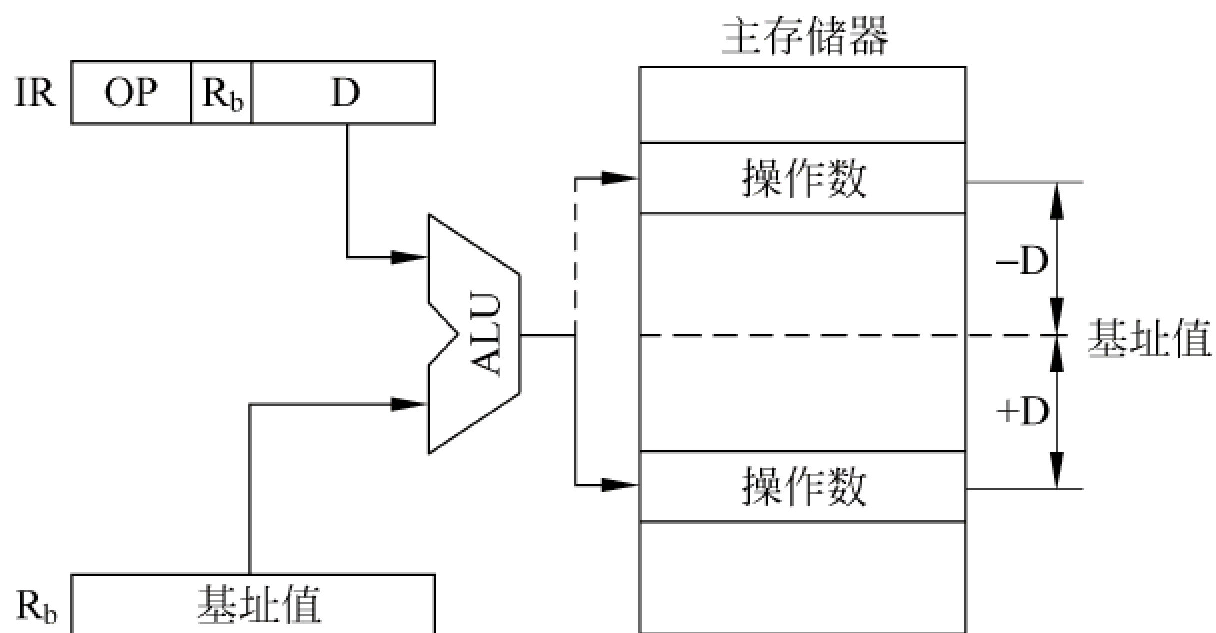


图3-8　基址寻址过程

基址寻址原是大型计算机采用的一种技术,用来将用户的逻辑地址(用户编程时使用的地址)转化成主存的物理地址(程序在主存中的实际地址)。

基址寻址和变址寻址在形成有效地址时所用的算法是相同的,而且在一些计算机中,这两种寻址方式都是由同样的硬件来实现的。但是,它们两者实际上是有区别的。一般来说,变址寻址中变址寄存器提供修改量(可变的),而指令中提供基准值(固定的);基址寻址中基址寄存器提供基准值(固定的),而指令中提供位移量(可变的)。这两种寻址方式应用的场合也不同,变址寻址是面向用户的,用于访问字符串、向量和数组等成批数据;而基址寻址面向系统,主要用于逻辑地址和物理地址的变换,用以解决程序在主存中的再定位和扩大寻址空间等问题。在某些大型机中,基址寄存器只能由特权指令来管理,用户指令无权操作和修改。在某些小、微型计算机中,基址寻址和变址寻址实际上是合二为一的。

8. 相对寻址

相对寻址是基址寻址的一种变通,由程序计数器(PC)提供基准地址,指令中的地址码字段作为位移量D,两者相加后得到操作数的有效地址,即 $EA=(PC)+D$。位移量指出的是操作数和现行指令之间的相对位置,如图3-9所示。

这种寻址方式有如下两个特点:

① 操作数的地址不是固定的,它随着PC值的变化而变化,并且与指令地址之间总是相差一个固定值。当指令地址变换时,由于其位移量不变,使得操作数与指令在可用的存储区

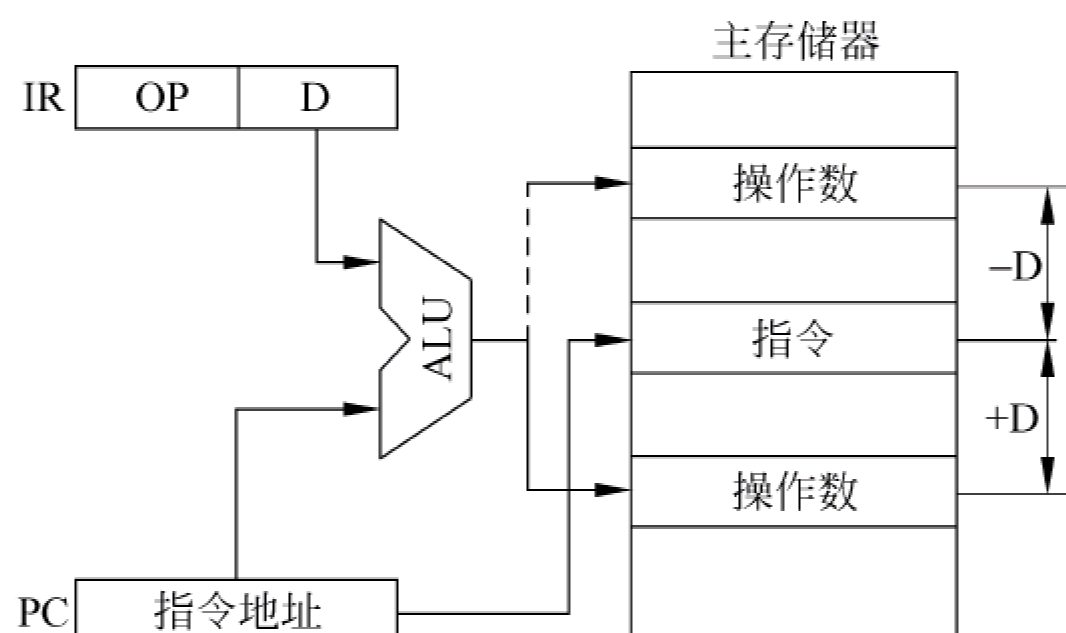


图 3-9　相对寻址过程

内一起移动,所以仍能保证程序的正确执行。采用 PC 相对寻址方式编写的程序可在主存中任意浮动,它放在主存的任何地方所执行的效果都是一样的。

② 对于指令地址而言,操作数地址可能在指令地址之前或之后,因此指令中给出的位移量可负、可正,通常用补码表示。如果位移量为 n 位,则相对寻址的寻址范围为:

$$(PC)-2^{(n-1)} \sim (PC)+2^{(n-1)}-1$$

注意:有些计算机是以当前指令地址为基准的,有些计算机是以下一条指令地址为基准的。这是因为有的机器是在当前指令执行完时,才将 PC 的内容加 1(或加增量);而有的机器是在取出当前指令后立即将 PC 的内容加 1(或加增量),使之变成下一条指令的地址。后一种方法将使位移量的计算变得比较复杂,特别是对于变字长指令更加麻烦。不过在实际应用时,位移量是由汇编程序自动形成的,程序员并不需要特别关注。

9. 页面寻址

页面寻址相当于将整个主存空间分成若干个大小相同的区,每个区称为一页,每页有若干个主存单元。例如,1 个 64KB 的存储器被划分为 256 个页面,每个页面中有 256 个字节,如图 3-10(a)所示。每页都有自己的编号,称为页面地址;页面内的每个主存单元也有自己的编号,称为页内地址。这样,存储器的有效地址就被分为两部分:前部为页面地址(在此例中占 8 位),后部为页内地址(也占 8 位)。页内地址由指令的地址码部分自动直接提供,它与页面地址通过简单的拼装连接就可得到有效地址,无须进行计算,因此寻址迅速。根据页面地址的来源不同,页面寻址又可以分成以下 3 种不同的方式。

① 基页寻址。基页地址又称零页寻址。由于页面地址全等于 0,所以有效地址 EA=0//A(//在这里表示简单拼接),操作数 S 在零页面中,如图 3-10(b)所示。基页寻址实际上就是直接寻址。

② 当前页寻址。页面地址就等于程序计数器(PC)的高位部分的内容,所以有效地址 $EA=(PC)_H//A$,操作数 S 与指令本身处于同一页面中,如图 3-10(c)所示。

③ 页寄存器寻址。页面地址取自页寄存器,与形式地址相拼接形成有效地址,如图 3-10(d)所示。

前两种方式因不需要页寄存器,所以用得较多些。有些计算机在指令格式中设置了一个页面标志位(Z/C)。Z/C=0,表示 0 页寻址;Z/C=1,表示当前页寻址。

怎样才能知道一条指令所采用的是什么寻址方式呢?为了能区分出各种不同的寻址方

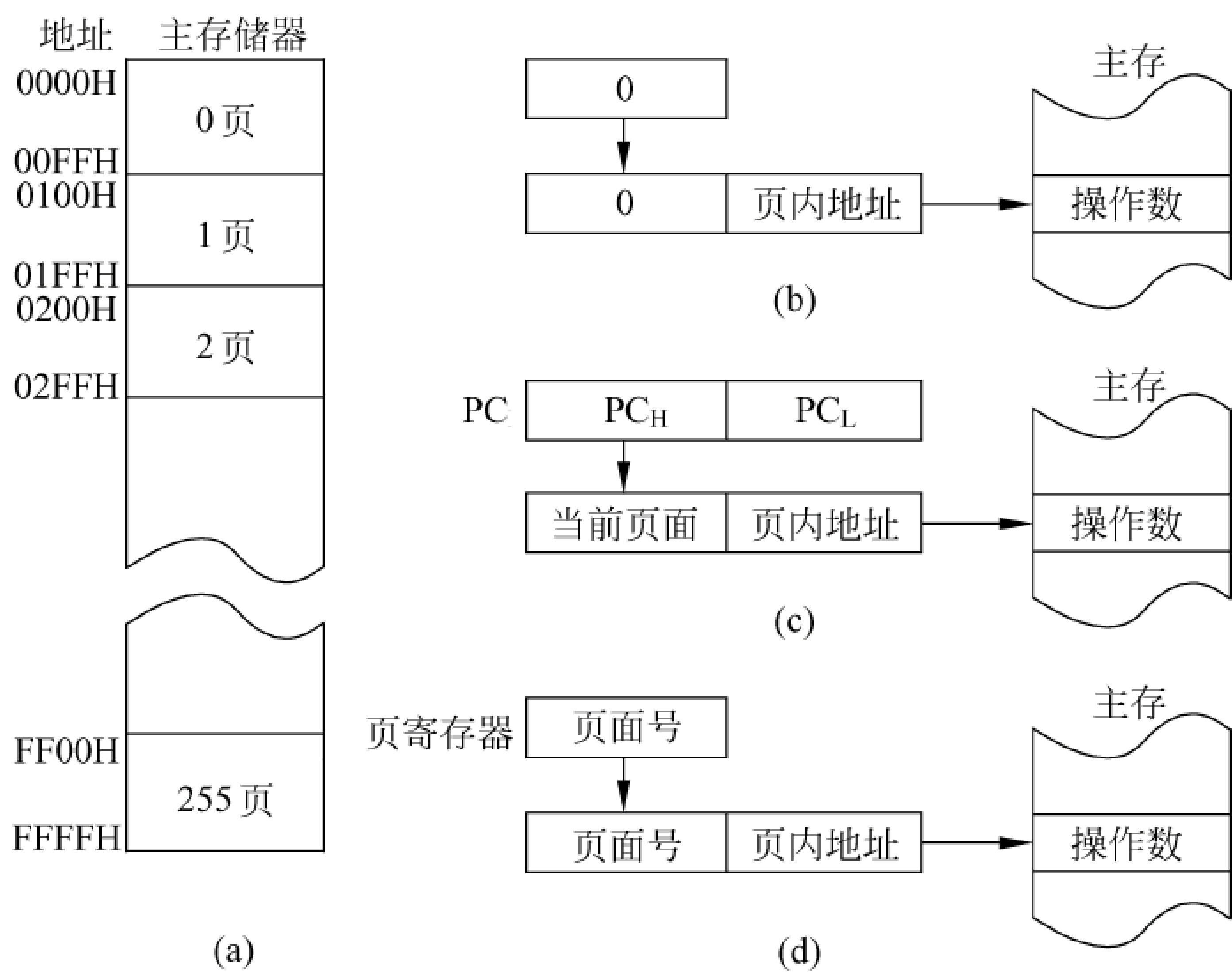


图 3-10　页面寻址

式，必须在指令中给出标识。标识的方式通常有两种：显式和隐式。显式的方法就是在指令中设置专门的寻址方式字段，用二进制编码来表明寻址方式类型，如图 3-11(a)所示；隐式的方式是由指令的操作码字段说明指令格式并隐含约定寻址方式，如图 3-11(b)所示。

注意：一条指令若有两个或两个以上的地址码时，各地址码可采用不同的寻址方式。例如，源地址采用一种寻址方式，而目的地址采用另一种寻址方式。

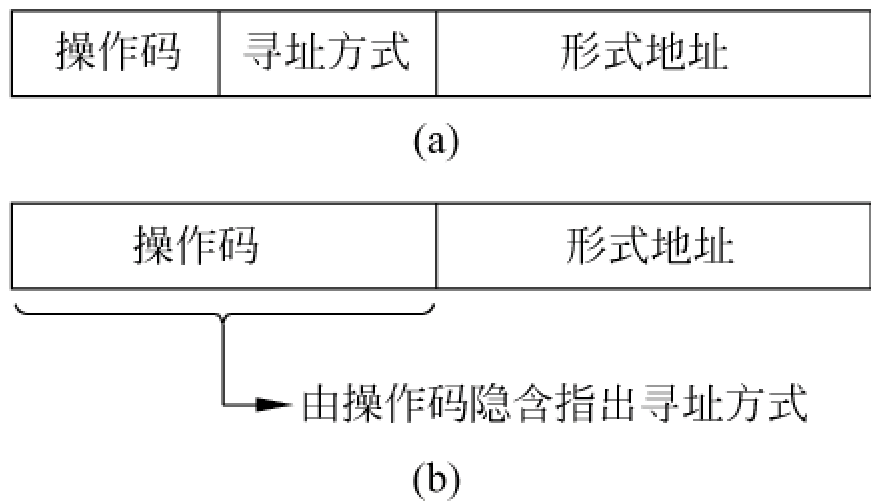


图 3-11　指令中寻址方式的表示

3.2.4　变型或组合寻址方式

前面介绍了 9 种常用的基本寻址方式，其他的寻址方式则是这 9 种寻址方式的变型或组合。

1. 自增型寄存器间址和自减型寄存器间址

这两种寻址方式实际上都是寄存器间接寻址方式的变型，通用寄存器在这里作为自动变址寄存器。

(1) 自增寻址

在自增寻址时，寄存器 R_i 的内容是有效地址，按照这个有效地址从主存中取数以后，寄存器的内容自动增量修改。在字节编址的计算机中，若指向下一个字节，寄存器的内容加 1；若指向下一个字(假设字长 16 位)，寄存器的内容加 2，如图 3-12(a)所示。

寻址操作的含义为：$EA=(R_i)$，$R_i \leftarrow (R_i)+d$。其中，EA 为有效地址，d 为修改量，通常记作$(R_i)+$，加号在括号之后，形象地表示先操作后修改。

(2) 自减寻址

自减寻址是先对寄存器 R_i 的内容自动减量修改(减1或减2),修改之后的内容才是操作数的有效地址,据此可到主存中取出操作数。图 3-12(b)给出了自减寻址过程示意图。

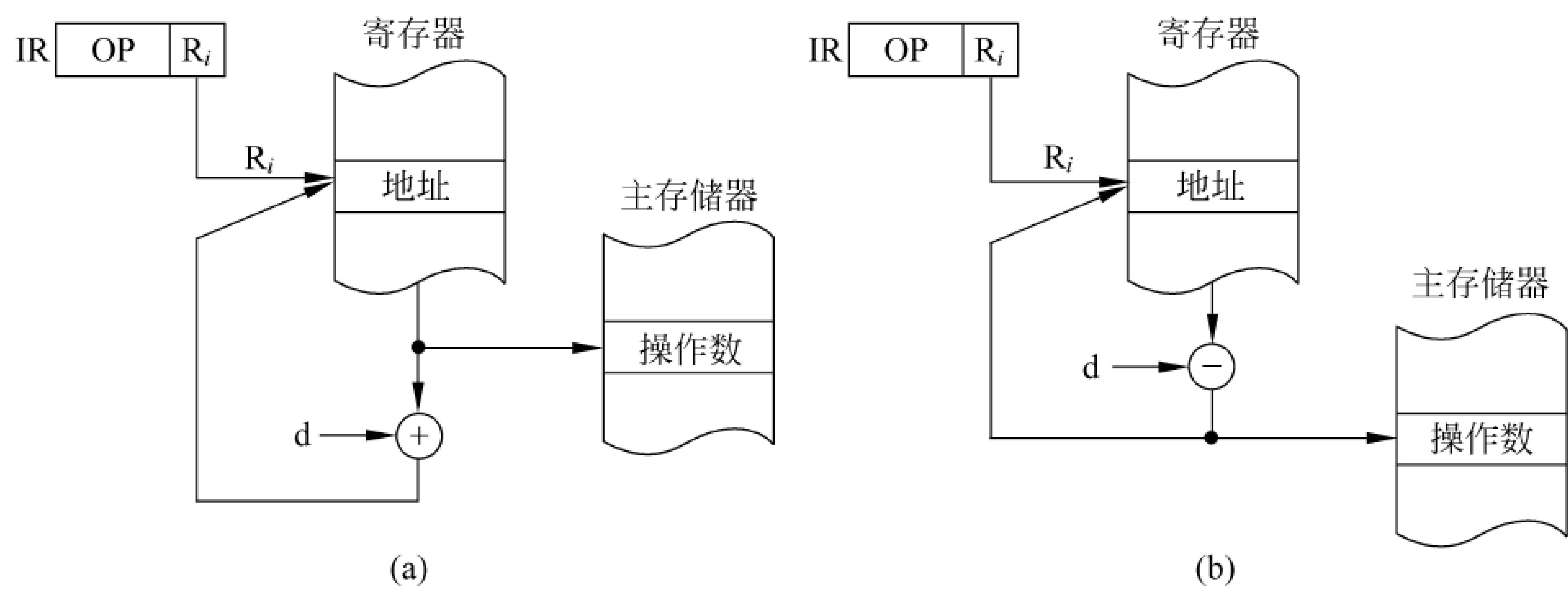


图 3-12　自增/自减寻址方式

寻址操作的含义为:$R_i \leftarrow (R_i) - d$,$EA = (R_i)$,通常记作 $-(R_i)$,减号在括号之前,形象地表示先修改后操作。自减寻址和自增寻址一起,可以使任何一个寄存器作为堆栈指针。

采用自增/减寻址最灵活的当属 MC68000 机,它具有字节、字、双字的自动增/减寻址方式。

2. 扩展变址方式

把变址和间址两种寻址方式结合起来就成为扩展变址方式,按寻址方式操作的先后顺序,有前变址和后变址两种寻址方式。

(1) 先变址后间址(前变址寻址方式)

先进行变址运算,其运算结果作为间接地址,间接地址指出的单元的内容才是有效地址。所以,有效地址 $EA = ((R_x) + A)$,操作数 $S = (((R_x) + A))$。其寻址过程如图 3-13(a)所示。

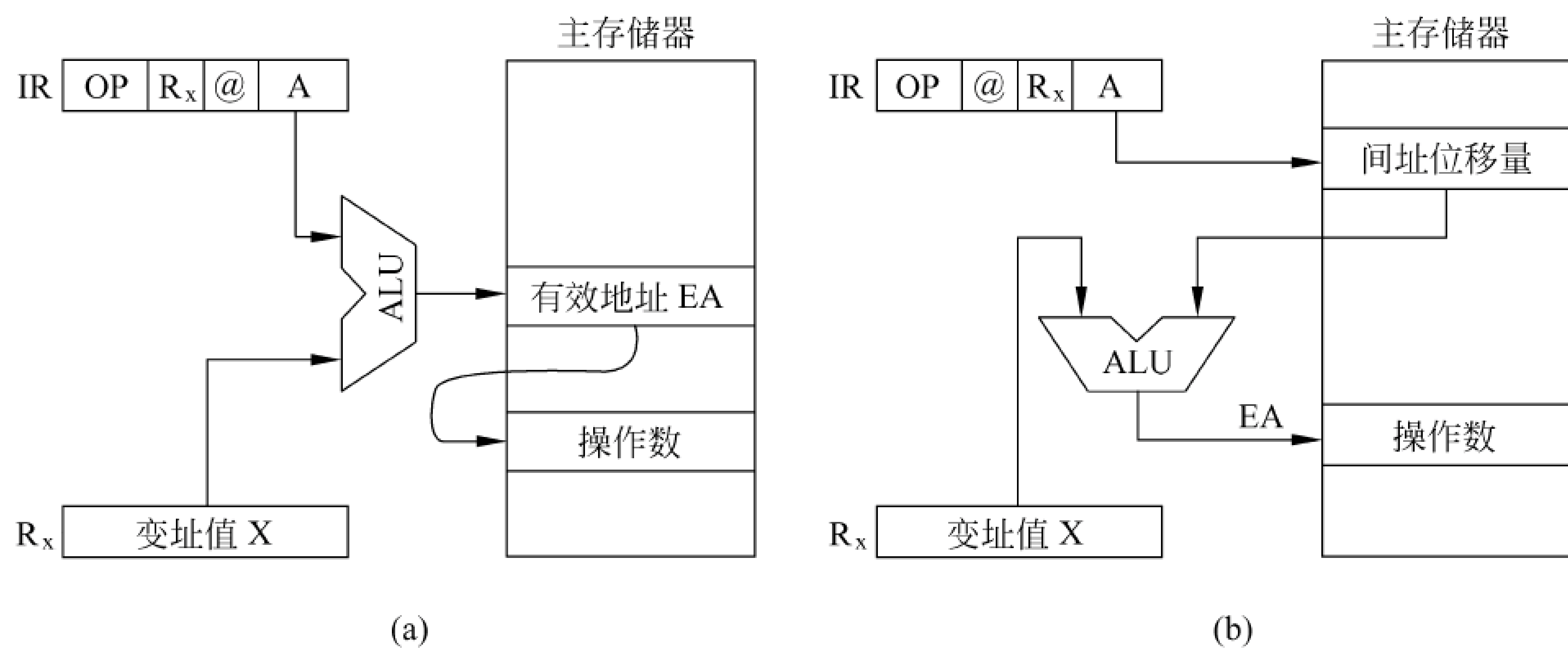


图 3-13　前/后变址寻址方式

（2）先间址后变址（后变址寻址方式）

将指令中的地址码先进行一次间接寻址，然后再与变址值进行运算，从而得到一个有效地址。所以，有效地址 $EA=(R_x)+(A)$，操作数 $S=((R_x)+(A))$。其寻址过程如图 3-13(b)所示。

3. 基址变址寻址

基址变址寻址是最灵活的一种寻址方式，此时有效地址是由基址寄存器中的值、变址寄存器中的值和位移量三者相加求得的。在这 3 项中，除位移量在指令一旦确定后就不能再修改以外，基址和变址寄存器中的内容都可以改变。

$$EA=(R_b)+(R_x)+D$$

其中，R_b 为基址寄存器，R_x 为变址寄存器，D 为位移量。

IBM 370 机中就有这种寻址方式。实际上，R_b 和 R_x 并不单独存在，通常借用 16 个通用寄存器中的 15 个（0 寄存器除外）来作为 R_b 或 R_x。上式 3 项中的任何一项都可以缺省。

基址变址寻址方式在 Intel 80x86 中是最基本的寻址方式，其他多种方式可由它派生出来。基址寄存器（BX 或 BP）、变址寄存器（SI 或 DI）及位移量都可以缺省，位移量允许是 8 位或 16 位的带符号数。

$$EA=\left\{\begin{matrix}(BX)\\(BP)\end{matrix}\right\}+\left\{\begin{matrix}(SI)\\(DI)\end{matrix}\right\}+\text{位移量}$$

3.3 堆栈与堆栈操作

堆栈是一种按特定顺序进行存取的存储区，这种特定顺序可归结为“后进先出（LIFO）”或“先进后出（FILO）”。在一般计算机中，堆栈主要用来暂存中断断点、子程序调用时的返回地址、状态标志及现场信息等，也可用于子程序调用时参数的传递。

3.3.1 堆栈结构

堆栈区通常是主存储器中指定的一个区域，也可以专门设置一个小而快的存储器作为堆栈区。在堆栈容量很小的情况下，还可以用一组寄存器来构成堆栈。

1. 寄存器堆栈

有些计算机中用一组专门的寄存器构成寄存器堆栈，又称为硬堆栈。这种堆栈的栈顶是固定的，寄存器组中各寄存器是相互连接的，它们之间具有对应位自动推移的功能，即可将一个寄存器的内容推移到相邻的另一个寄存器中，如图 3-14 所示。在执行压入操作（进栈）时，一个压入信号将使所有寄存器的内容依次向下推移一个位置，即寄存器 i 的内容被传送到 $i+1$，同时一个 n 位的数据被压入栈顶（寄存器 0）。在执行弹出操作（出栈）时，一个弹出信号将把所有寄存器的内容依次向上推移一个位置，即寄存器 i 的内容被传送到寄存器 $i-1$，栈顶（寄存器 0）的内容被弹出。

从图 3-14 可看出，上述堆栈中最多只能压入 k 个数据，否则将丢失信息。这种堆栈的工作过程很像子弹夹的弹仓，由于栈顶位置固定，故不必设置堆栈的栈顶指针。

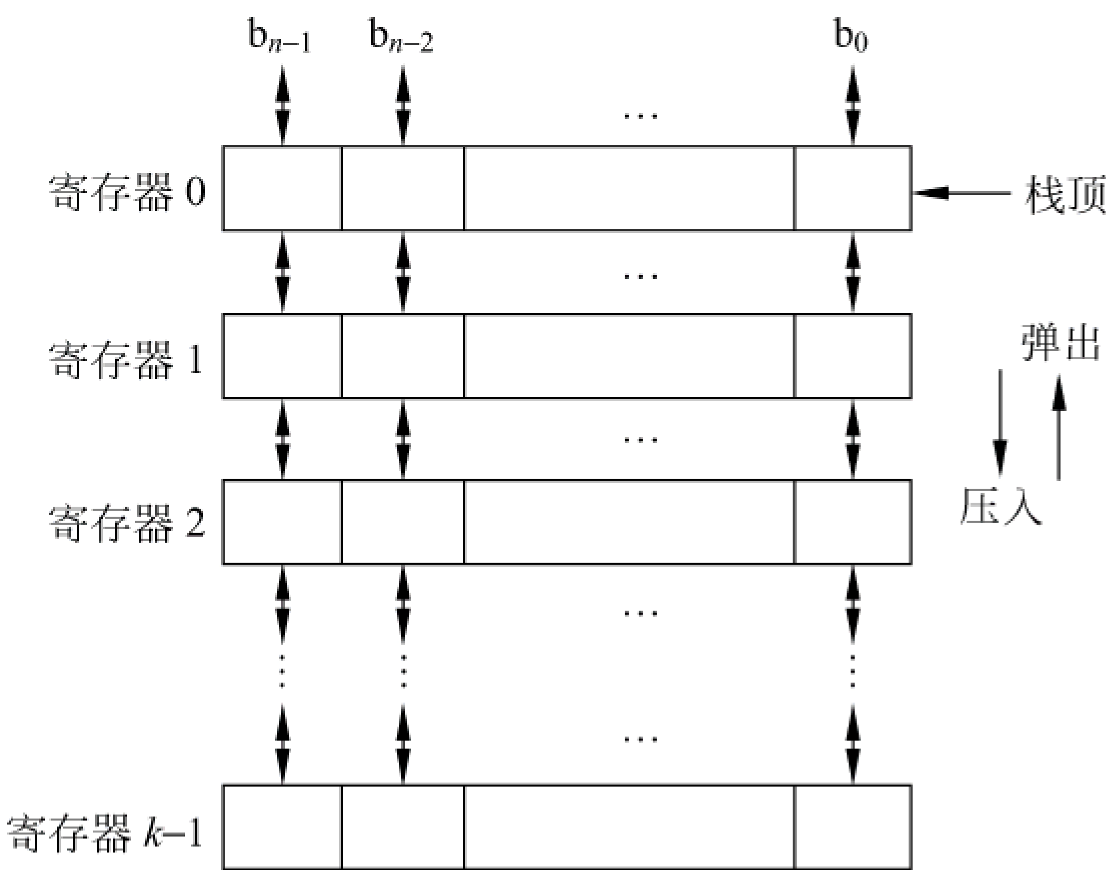


图 3-14　寄存器堆栈结构

2. 存储器堆栈

寄存器堆栈的成本比较高,不适于作大容量的堆栈,而从主存中划出一段区域来作为堆栈是最合算且最常用的方法。这种堆栈又称为软堆栈,堆栈的大小可变,栈底固定,栈顶浮动,故需要一个专门的硬件寄存器作为堆栈栈顶指针,简称栈指针(SP)。栈指针所指定的存储单元就是堆栈的栈顶。存储器堆栈又可分为两种:自底向上生成堆栈和自顶向下生成堆栈。假设栈指针始终指向栈顶的满单元,且压入和弹出的数据为一个字节。

(1) 自底向上生成(向低地址方向生成)堆栈

这种堆栈的栈底地址大于栈顶地址,如图 3-15 所示。因此,进栈时,堆栈指针 SP 的内容需要先自动减 1,然后再将数据压入堆栈;出栈时,需要先将堆栈中的数据弹出,然后 SP 的内容再自动加 1。进、出栈的过程可描述如下。

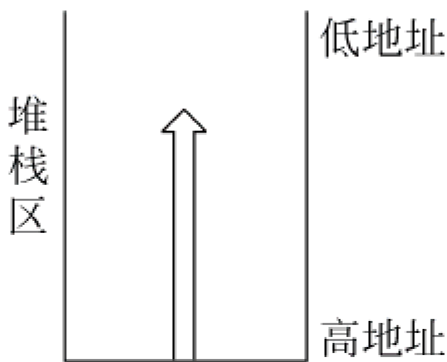


图 3-15　存储器堆栈结构

进栈:

(SP)－1→SP　　;修改栈指针
(A)→(SP)　　;将 A 中的内容压入栈顶单元

出栈:

((SP))→A　　;将栈顶单元内容弹出送入 A 中
(SP)＋1→SP　　;修改栈指针

其中,A 为寄存器或主存单元地址;(SP)表示堆栈指针的内容,即栈顶单元地址;((SP))表示栈顶单元的内容。

(2) 自顶向下生成(向高地址方向生成)堆栈

这种堆栈与自底向上堆栈正好相反,它的栈底地址小于栈顶地址。进栈时,先令(SP)＋1→SP,然后再压入数据;出栈时,先将数据弹出,然后(SP)－1→SP。

软堆栈的容量可以很大,而且可以在整个主存中浮动,但是速度比较慢,每访问一次堆栈实际就是访问一次主存。在一些大型的计算机系统中,希望堆栈的容量大、速度快,故将

前述两种堆栈组合起来构成软、硬结合的堆栈。在这样的堆栈中，一般压入、弹出操作在小容量的硬堆栈中进行，这样可保证访问速度快。当硬堆栈已满之后，每向硬堆栈压入一个数据，总是将其栈底寄存器中的数据压入软堆栈中，使堆栈总容量有效扩大；同样，数据出栈时，不断将软堆栈中栈顶的内容上移至硬堆栈的栈底寄存器中。显然它集中了硬堆栈速度快、软堆栈容量大的优点，只是在控制上稍复杂些，但这是完全可以实现的。

3.3.2 堆栈操作

堆栈操作既不是在堆栈中移动它所存储的内容，也不是把已存储在栈中的内容从栈中抹掉，而是通过调整堆栈指针而给出新的栈顶位置，以便对位于栈顶位置的数据进行操作。

在一般计算机中，堆栈主要用来暂存中断断点、子程序调用时的返回地址、状态标志及现场信息等，也可用于子程序调用时参数的传递，所以用于访问堆栈的指令只有进栈(压入)和出栈(弹出)两种。

在堆栈计算机(如 HP-3000 和 B5000 机等)中，算术逻辑类指令中没有地址码字段，故称为零地址指令。参加运算的两个操作数隐含地从堆栈顶部弹出，送到运算器中进行运算，运算的结果再隐含地压入堆栈。如果将算术表达式改写为逆波兰表达式，用零地址指令进行运算是十分方便的。例如，有算术表达式 $a\times b+c\div d$，运算结果送给 X，这个算术表达式可以用逆波兰法表示为 $ab\times cd\div+$。现在用零地址指令和一地址指令对该算式编程，并利用堆栈完成运算。假设堆栈采用自底向上生成方式，用大写字母 A 表示数据 a 的地址，其他依次类推，其程序段为：

```
PUSH A        ;数据 a 压入堆栈
PUSH B        ;数据 b 压入堆栈
MUL           ;完成 a×b
PUSH C        ;数据 c 压入堆栈
PUSH D        ;数据 d 压入堆栈
DIV           ;完成 c÷d
ADD           ;完成 a×b+c÷d
POP X         ;结果存入 X 单元
```

注意：执行一条零地址的双操作数运算指令，如果是软堆栈，则需要访问 4 次主存；如果是硬堆栈，则只需要访问一次主存。

3.4 指令类型

一台计算机的指令系统可以有上百条指令，这些指令按其功能可以分成几种类型，下面分别介绍。

3.4.1 数据传送类指令

数据传送类指令是最基本的指令类型，主要用于实现寄存器与寄存器之间、寄存器与主存单元之间以及两个主存单元之间的数据传送。数据传送类指令又可以细分为下列几种。

1. 一般传送指令

一般传送指令具有数据复制的性质，即数据从源地址传送到目的地址，而源地址中的内容保持不变。一般传送类指令常用助记符 MOV 表示，根据数据传送的源和目的的不同，又可分为以下几种传递方式：

① 主存单元之间的传送。

② 从主存单元传送到寄存器。在有些计算机中，该指令用助记符 LOAD(取数指令)表示。

③ 从寄存器传送到主存单元。在有些计算机里，该指令用助记符 STORE(存数指令)表示。

④ 寄存器之间的传送。

2. 堆栈操作指令

堆栈指令实际上是一种特殊的数据传送指令，分为进栈(PUSH)和出栈(POP)两种，在程序中它们往往是成对出现的。

如果堆栈是主存的一个特定区域，那么对堆栈的操作也就是对存储器的操作。

3. 数据交换指令

前述的传送都是单方向的。然而，数据传送也可以是双方向的，即将源操作数与目的操作数(一个字节或一个字)相互交换位置。

3.4.2 运算类指令

1. 算术运算类指令

算术运算指令主要用于定点和浮点运算。这类运算包括定点加、减、乘、除指令，浮点加、减、乘、除指令以及加1、减1、比较等，有些机器还有十进制算术运算指令。

绝大多数算术运算指令都会影响到状态标志位，通常的标志位有进位、溢出、全零、正负和奇偶等。

为了实现高精度的加减运算(双倍字长或多字长)，低位字(字节)加法运算所产生的进位(或减法运算所产生的借位)都存放在进位标志中；在高位字(字节)加减运算时，应考虑低位字(字节)的进位(或借位)，因此，指令系统中除去普通的加、减指令外，一般都设置了带进位加指令和带借位减指令。

2. 逻辑运算类指令

一般计算机都具有与、或、非和异或等逻辑运算指令。这类指令在没有设置专门的位操作指令的计算机中常用于对数据字(字节)中某些位(一位或多位)进行操作，常见的应用如下。

(1) 按位测(位检查)

利用“与”指令可以屏蔽掉数据字(字节)中的某些位。通常让被检查数作为目的操作

数，屏蔽字作为源操作数，要检测某些位，可使屏蔽字的相应位为“1”，其余位为“0”，然后执行“与”指令，则可取出所要检查的位来。

（2）按位清（位清除）

利用“与”指令还可以使目的操作数的某些位置“0”。只要源操作数的相应位为“0”，其余位为“1”，然后执行“与”指令即可。

（3）按位置（位设置）

利用“或”指令可以使目的操作数的某些位置“1”。只要源操作数的相应位为“1”，其余位为“0”，然后执行“或”指令即可。

（4）按位修改

利用“异或”指令可以修改目的操作数的某些位，只要源操作数的相应位为“1”，其余位为“0”，“异或”之后就达到了修改这些位的目的（因为 $A\oplus1=\overline{A}$，$A\oplus0=A$）。

（5）判符合

若两数相符合，其“异或”之后的结果必定为全“0”。

3. 移位类指令

移位指令分为算术移位、逻辑移位和循环移位 3 类，它们又可分为左移和右移两种。

（1）算术移位

算术移位的对象是带符号数，在移位过程中必须保持操作数的符号不变。当左移一位时，如不产生溢出，则数值乘以 2；而右移一位时，如不考虑因移出舍去的末位尾数，则数值除以 2，如图 3-16(a)所示。

（2）逻辑移位

逻辑移位的对象是无符号数，因此移位时不必考虑符号问题，如图 3-16(b)所示。从图中可以看出，逻辑左移指令和算术左移指令移位操作过程完全相同，这是因为正确的算术左移（不产生溢出时）与逻辑左移结果相同。

（3）循环移位

循环移位按是否与进位位一起循环又分为两种：小循环（不带进位循环），如图 3-16(c)所示；大循环（带进位循环），如图 3-16(d)所示。

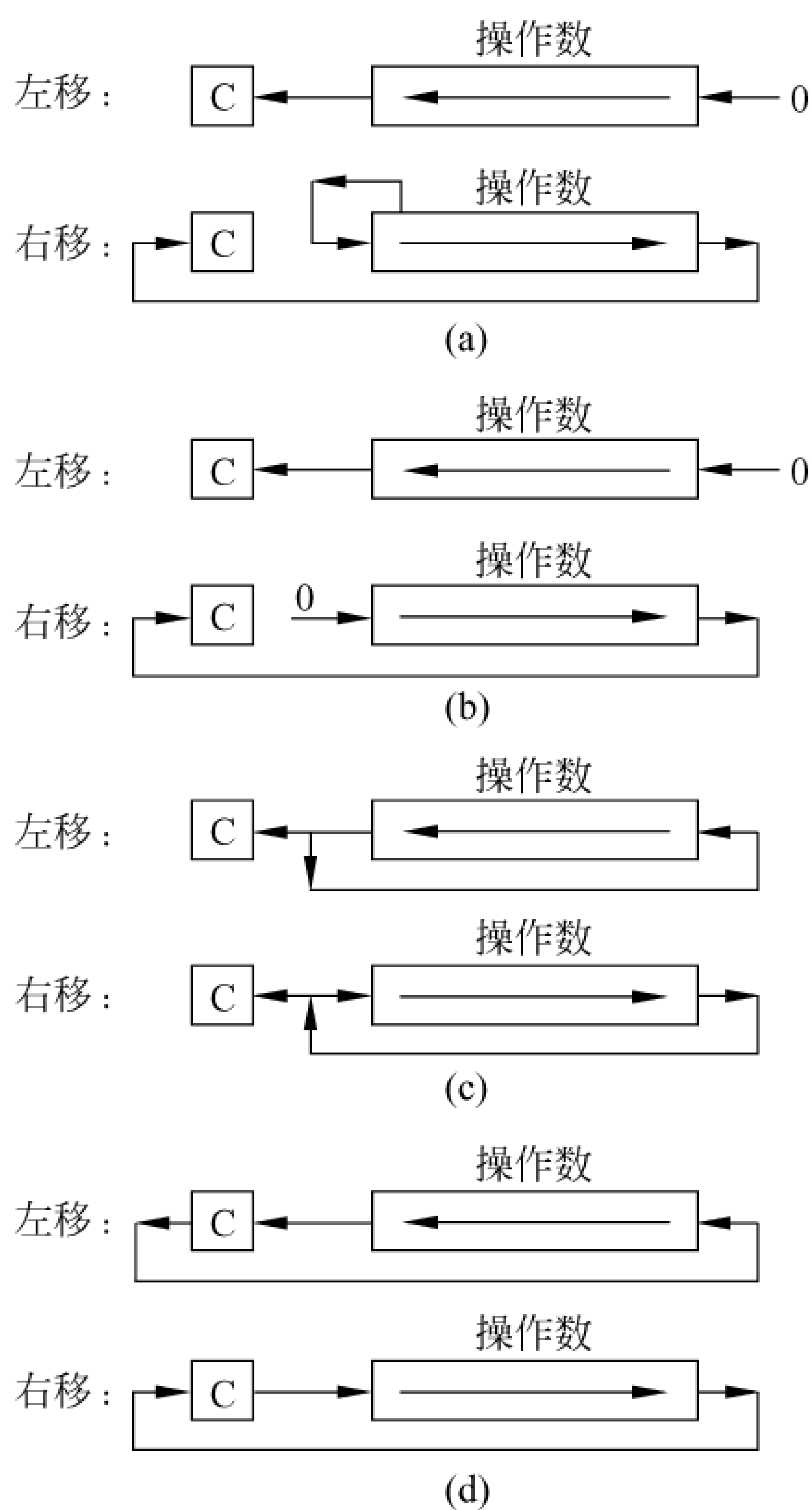


图 3-16　移位操作过程

3.4.3　程序控制类指令

程序控制类指令用于控制程序的执行顺序，并使程序具有测试、分析与判断的能力。因此，它们是指令系统中一组非常重要的指令，主要包括转移指令、子程序调用和返回指令等。

1. 转移指令

在程序执行过程中,通常采用转移指令来改变程序的执行顺序。转移指令又分无条件转移和条件转移两种:

① 无条件转移又称必转,它在执行时将改变程序的常规执行顺序,不受任何条件的约束,直接把程序转向该指令指出的新的位置并执行,其助记符一般为JMP。

② 条件转移必须受到条件的约束,若满足指令所规定条件,则程序转移;否则,程序仍顺序执行。条件转移指令主要用于程序的分支,当程序执行到某处时,要在两个分支中选择一支,这就需要根据某些测试条件做出判断。

无论是条件转移还是无条件转移都需要给出转移地址。若采用相对寻址方式,则转移地址为当前指令地址(即PC的值)和指令中给出的位移量之和,即(PC)+位移量→PC;若采用绝对寻址方式,则转移地址由指令的地址码字段直接给出,即A→PC。

条件转移指令采用相对寻址方式,通常位移量只有一个字节,这样转移范围只能在离当前PC的−128~+127个字节之内,在32位的80x86中,允许采用多字节表示位移量,此时转移范围可以超出原来的−128~+127。

转移的条件以某些标志位或这些标志位的逻辑运算作为依据,根据单个标志位的条件转移指令的转移条件是上次运算结果的某些标志,如进位标志、结果为零标志、结果溢出标志等,而用于无符号数和带符号数的条件转移指令的转移条件则是上述标志位逻辑运算的结果。

无符号数之间大小比较后的条件转移指令和带符号数之间的大小比较后的条件转移指令有很大不同。带符号数间的次序关系称为大于(G)、等于(E)和小于(L);无符号数间的次序关系称为高于(A)、等于(E)和低于(B)。

2. 子程序调用指令

子程序是一组可以公用的指令序列,只要知道子程序的入口地址就能调用它。通常把一些需要重复使用并能独立完成某种特定功能的程序单独编成子程序,在需要时由主程序调用它们,这样做既简化了程序设计,又节省了存储空间。

主程序和子程序是相对的概念,调用其他程序的程序是主程序,而被其他程序调用的程序是子程序。子程序允许嵌套,即程序A调用程序B,程序B又调用程序C,程序C再调用程序D……这个过程又称为多重转子。其中,程序B对于程序A来说是子程序,而程序B对于程序C来说是主程序。另外,子程序还允许自己调用自己,即子程序递归。

从主程序转向子程序的指令称为子程序调用指令,简称转子指令,其助记符一般为CALL。转子指令安排在主程序中需要调用子程序的地方,转子指令是一地址指令。

转子指令和转移指令都可以改变程序的执行顺序,但事实上两者存在着很大的差别:

① 转移指令使程序转移到新的地址后继续执行指令,不存在返回的问题,所以没有返回地址;而转子指令要考虑返回问题,所以必须以某种方式保存返回地址,以便返回时能找到原来的位置。

② 转移指令用于实现同一程序内的转移;而转子指令转去执行一段子程序,实现的是不同程序之间的转移。

返回地址是转子指令的下一条指令的地址，保存返回地址的方法有多种：

① 用子程序的第一个字单元存放返回地址。转子指令把返回地址存放在子程序的第一个字单元中，子程序从第二个字单元开始执行。返回时将第一个字单元地址作为间接地址，采用间址方式返回主程序。这种方法可以实现多重转子，但不能实现递归循环，如Cyber70机采用的就是这种方法。

② 用寄存器存放返回地址。转子指令先把返回地址放到某一个寄存器中，再由子程序将寄存器中的内容转移到另一个安全的地方，如主存的某个区域。这是一种较为安全的方法，可以实现子程序的递归循环。IBM 370机采用的就是这种方法，这种方法相对增加了子程序的复杂程度。

③ 用堆栈保存返回地址。不管是多重转子还是子程序递归，最后存放的返回地址总是最先被使用的，堆栈的后进先出存取原则正好支持实现多重转子和递归循环，而且也不增加子程序的复杂程度。这是应用最为广泛的方法。例如，PDP-11、VAX-11、Intel 80x86机等均采用这种方法。

3. 返回指令

从子程序转向主程序的指令称为返回指令，其助记符一般为RET，子程序的最后一条指令一定是返回指令。返回地址存放的位置决定了返回指令的格式，通常返回地址保存在堆栈中，所以返回指令常是零地址指令。

转子和返回指令也可以是带条件的，条件转子和条件返回与前述条件转移的条件是相同的。

3.4.4 输入输出类指令

输入输出(I/O)类指令用来实现主机与外部设备之间的信息交换，包括输入输出数据、主机向外设发控制命令或外设向主机报告工作状态等。从广义的角度看，I/O指令可以归入数据传送类。各种不同计算机的I/O指令差别很大，通常有两种编址方式：独立编址方式和统一编址方式。

1. 独立编址的I/O指令

独立编址方式使用专门的输入输出指令(IN/OUT)。以主机为基准，信息由外设传送给主机称为输入，反之称为输出。指令中应给出外部设备编号(端口地址)。这些端口地址与主存地址无关，是另一个独立的地址空间。80x86的I/O指令采用的就是独立编址方式。

2. 统一编址的I/O指令

所谓统一编址，就是把外设寄存器和主存单元统一编址。在这种方式下，不需要专门的I/O指令，就用一般的数据传送类指令来实现I/O操作。一个外部设备通常至少有两个寄存器：数据寄存器以及命令与状态寄存器。每个外设寄存器都可以由分配给它们的唯一的主存地址来识别，主机可以像访问主存一样去访问外部设备的寄存器。PDP-11机采用的就是统一编址方式，它把最高4KB主存地址作为外设寄存器的地址。

这两种编址方式各有优缺点，它们的比较如表3-2所示。

表 3-2 两种编址方式比较

优缺点	独立编址方式	统一编址方式
优点	I/O 指令和访存指令容易区分,外设地址线少,译码简单,主存空间不会减少	总线结构简单,全部访存类指令都可用于控制外设,可直接对外设寄存器进行各种运算
缺点	控制线增加了 I/O Read 和 I/O Write 信号	占用主存一部分地址,缩小了可用的主存空间

3.4.5 80x86 指令系统举例

从 80386 开始的微处理器属于 IA-32 结构,下面以部分常用指令为例介绍 80x86 的指令系统。

1. MOV 指令

这是一种形式最简单、使用最频繁的指令,它可以实现寄存器与寄存器之间、寄存器与主存单元之间的数据传送,也可以将立即数传送到寄存器。

MOV 指令的传送通常以字节、字、双字为单位,应当保持数据宽度一致,否则需要使用汇编语言的指示符。

注意:*MOV 指令的源操作数和目的操作数中,必须有一个在寄存器中,不允许用于两个主存单元之间的数据传送,并且不能向代码寄存器(CS)和堆栈寄存器(SS)传送数据。*

2. PUSH/POP 指令

进栈指令(PUSH)可以分别将寄存器、主存、段寄存器、状态标志寄存器和全部寄存器(80386 以上)的内容或立即数压入到堆栈中。出栈指令(POP)则弹出保存的数据,但不能从堆栈中弹出数据至立即数,也不能将数据弹出至代码段寄存器。

堆栈位置由堆栈寄存器(SS)和堆栈指针(SP)规定。在 80x86 中,堆栈操作都是字(16位)操作,同时还限定压入数据的来源和弹出数据的去向不能是主存单元。

3. 加、减和比较指令

加法/减法指令(ADD/SUB)所需的操作数可以在寄存器、主存中,也可以是立即数。加 1 或减 1 指令(INC/DEC)的操作数在寄存器中。

比较指令(CMP)是减法指令的一个特殊变化,仍是进行两数相减的运算,但结果不回送,即不保留"差"。比较指令的功能在于不破坏原来的两个操作数,而仅设置相应的标志位。

为了实现高精度的加减运算(双倍字长或多字长),除去普通的加、减指令外还设置了带进位加指令(ADC)和带借位减指令(SBB)。

4. 乘法、除法指令

乘法允许进行字节、字或双字运算,它们可以是带符号的(IMUL)或无符号的(MUL)整数。被乘数分别存放在 AL、AX 或 EAX 中,乘数可在其他数据寄存器中,乘积是双倍宽的数据,字节乘法的积存放在 AX 中,字乘法的积存放在 DX(高 16 位数据)和 AX(低 16 位

数据)中,双字乘法的积存放在 EDX(高 32 位数据)和 EAX(低 32 位数据)中。

除法也可以进行字节、字或双字运算。它们也可以是带符号的(IDIV)或无符号的(DIV)整数。被除数总是双倍宽的数据。对于 8 位的除数,被乘数存放在 AX 中;对于 16 位的除数,被除数存放在 DX 和 AX 中,对于 32 位的除数,被除数存放在 EDX 和 EAX 中。

5. BCD 运算和 ASCII 运算

十进制运算调整指令(DAA)置于 ADD 或 ADC 指令之后,将加法运算的结果调整为 BCD 数的结果。由于 DAA 指令只作用于 AL 寄存器,因此这种运算每次只能做 8 位加法。

十进制运算调整指令(DAS)置于 SUB 或 SBB 指令之后,将减法运算的结果调整为 BCD 数的结果。

ASCII 算术运算指令作用于 ASCII 码数字。AAA、AAM、AAS 分别在加法、乘法、减法之后进行调整,AAD 在除法之前进行调整。

6. 基本逻辑指令

基本逻辑指令包括与(AND)、或(OR)、异或(XOR)、非(NOT)和测试(TEST)指令,它们允许进行字节、字或双字运算。

这些指令主要用于清零和屏蔽寄存器某些位的内容,其操作会影响到某些标志位。例如,"XOR AX,AX"指令可以对 AX 清零,还可以清除进位位和影响到 SF、ZF、PF 标志位。

TEST 指令实现 AND 的操作,但不改变目的操作数,仅仅影响标志寄存器的标志位。

7. 位测试指令

80386 以上的微处理器增加了位测试指令 BT、BTC、BTR 和 BTS。测试以后,将测试结果装入进位标志位,后 3 条指令还会改变被测试位。

8. 移位与循环指令

移位指令分为算术移位指令 SAL(左移)、SAR(右移)和逻辑移位指令 SHL(左移)、SHR(右移)。左移将操作数的最高位移入进位标志位,最低位补 0;右移将操作数的最低位移入进位标志位,对逻辑右移,最高位补 0,对算术右移,最高位(即带符号数的符号位)保持原值。

对于 80386 以上的微处理器,还有双精度移位指令 SHLD(左移)和 SHRD(右移),这两条指令有 3 个操作数,可以作用于两个 16 位或 32 位寄存器,或者作用于一个 16 位或 32 位主存单元与一个寄存器。

循环指令按是否与进位标志位一起循环又可细分为小循环(不带进位循环)和大循环(带进位循环)两种,同时具有左循环和右循环两种情况。故共有小循环左移(ROL)、小循环右移(ROR)、大循环左移(RCL)、大循环右移(RCR)4 种指令。

9. 转移控制指令

转移控制指令包括无条件转移指令、条件转移指令和程序循环指令。这些转移指令允许在执行程序过程中跳过一段程序,转到主存的任何部分去执行另一条指令。

无条件转移指令(JMP)不受任何条件的约束,跳转到由该指令指定的存储单元地址去执行另一条指令。

条件转移需要测试的标志位有进位标志位(CF)、零标志位(ZF)、符号标志位(SF)、溢出标志位(OF)和奇偶标志位(PF)等。这些标志位的组合,可以产生十几种条件转移指令。若条件满足,则转到指令指定的地址处;若条件不满足,则顺序执行程序的下一条指令。

程序循环指令有LOOP指令和LOOPE/LOOPNE指令。LOOP指令将CX/ECX减1并执行JNZ指令。如果CX/ECX不等于零,则转移到指定的地址去执行另外的指令。如果CX/ECX为零,则顺序执行下一条指令。LOOPE/LOOPNE是条件程序循环指令,以LOOPE指令(等于则循环)为例,如果CX/ECX不等于零且等于条件成立,则执行转移;如果不等于条件成立或CX/ECX减1后为零,则跳出循环。

10. 子程序调用和返回指令

子程序通过调用子程序指令(CALL)调用,通过返回指令(RET)返回。

在执行CALL指令时,返回地址(CS和IP寄存器的内容)被自动地压入堆栈保存。在执行RET指令时,自动地从堆栈中弹出返回地址送给CS和IP寄存器。

11. 输入输出指令

80x86微处理器中的I/O指令必须使用AL(8位)、AX(16位)或EAX(32位)进行传送,如表3-3所示。在I/O指令中可以直接给出I/O端口地址(Port),也可以由DX寄存器间接给出I/O端口地址。前者称为直接端口寻址,直接端口寻址最多只能寻址256个端口;后者称为间接端口寻址,间接端口寻址最多可以寻址65 536个端口。

表3-3　80x86微处理器的I/O指令

助记符	操作数	完成的操作
IN	Acc,Port	把指定端口中的内容输入到AL、AX或EAX中
IN	Acc,DX	把DX寄存器所指定的端口中的内容输入到AL、AX或EAX中
OUT	Port,Acc	将AL、AX或EAX的内容输出到指定端口中
OUT	DX,Acc	将AL、AX或EAX的内容输出到由DX寄存器所指定的端口中

3.5　指令系统的发展

不同类型的计算机有各具特色的指令系统,由于计算机的性能、机器结构和使用环境不同,指令系统的差异也是很大的。

3.5.1　x86架构的扩展指令集

目前,主流微机使用的指令系统都基于x86架构,为了提升处理器各方面的性能,Intel和AMD公司又各自开发了一些新的扩展指令集。扩展指令集中包含了处理器对多媒体、三维处理等方面的支持,能够提高处理器对这些方面处理的能力。

1. MMX 指令集

MMX(Multi Media eXtension,多媒体扩展)指令集是 Intel 公司为 Pentium 系列处理器所开发的一项多媒体指令增强技术。MMX 指令集中包括了 57 条多媒体指令,通过这些指令可以一次性处理多个数据,对视频、音频和图形数据处理特别有效。

2. SSE 指令集

SSE(Streaming SIMD Extension,流式 SIMD 扩展)也称为单指令多数据流(Single Instruction Multiple Data,SIMD)。SSE 指令集共有 70 条指令,其中包含提高三维图形运算效率的 50 条 SIMD 浮点运算指令、12 条 MMX 整数运算增强指令、8 条优化内存中的连续数据块传输指令。

3. 3DNow 指令集

3DNow 指令集最初由 AMD 公司推出,拥有 21 条扩展指令。3DNow 在整体上与 SSE 非常相似,但它与 SSE 的侧重点又有所不同,3DNow 指令集主要针对三维建模、坐标变换和效果渲染等三维数据的处理,在相应的软件配合下,可以大幅度提高处理器的三维处理性能。增强型 3DNow 共有 45 条指令,比 3DNow 又增加了 24 条指令。

4. SSE2 指令集

SSE2 包含了 144 条指令,分为 SSE 部分和 MMX 部分。SSE 部分主要负责处理浮点数,而 MMX 部分则专门计算整数。在指令处理速度保持不变的情况下,通过 SSE2 优化后的程序和软件运行速度也能够提高两倍。由于 SSE2 指令集与 MMX 指令集相兼容,因此被 MMX 优化过的程序很容易被 SSE2 再进行更深层次的优化,达到更好的运行效果。

5. SSE3 指令集

SSE3 是目前规模最小的指令集,它只有 13 条指令,被分为数据传输、数据处理、特殊处理、优化和超线程性能增强 5 个部分。其中,超线程性能增强是一种全新的指令集,它可以提升处理器的超线程的处理能力,大大简化超线程的数据处理过程,使处理器能够更加快速地进行并行数据处理。

6. SSSE3 指令集

SSSE3 是 Intel 公司针对 SSE3 指令集的一次额外扩充,有 32 条指令,进一步增强在多媒体、图形图像和 Internet 等方面的处理能力。

7. SSE4 指令集

SSE4 包含 47 条指令,主要针对向量绘图运算、三维游戏加速、视频编码加速及协同处理的加速。

8. SSE5 指令集

SSE5 是 AMD 为了打破 Intel 公司在处理器指令集的垄断地位而提出的,SSE5 加入了 100 余条新指令,其中最引人注目的就是三操作数指令及熔合乘法累积。三操作数指令增加了操作数的数量,使一条 x86 指令能处理 2 或 3 笔数据,从而把多个简单的指令整合到更高效的一个单独指令中,提高执行效率。熔合乘法累积技术可结合乘法与其他算法,保证只用一条指令就能完成迭代运算,从而简化代码,提高效率。

9. AVX 指令集

AVX(Advanced Vector eXtensions)是 Intel 公司的 SSE 延伸,支持三操作数指令。

10. FMA 指令集

FMA(Fused Multiply Accumulate)是 Intel 公司的 AVX 扩充指令集,支持熔合乘法累积。

3.5.2 从复杂指令系统到精简指令系统

指令系统的发展有两种截然不同的方向,一种是增强原有指令的功能,设置更为复杂的新指令实现软件功能的硬化;另一种是减少指令种类和简化指令功能,提高指令的执行速度。前者称为复杂指令系统,后者称为精简指令系统。

长期以来,计算机性能的提高往往是通过增加硬件的复杂性获得的,随着 VLSI 技术的迅速发展,硬件成本不断下降,软件成本不断上升,促使人们在指令系统中增加更多的指令和更复杂的指令,以适应不同应用领域的需要。这种基于复杂指令系统设计的计算机称为复杂指令系统计算机(Complex Instruction Set Computer,CISC)。CISC 的指令系统多达几百条指令,例如,Intel 80x86(IA-32)就是典型的 CISC,其中 Pentium 4 的指令条数已达到 500 多条(包括扩展的指令集)。

如此庞大的指令系统使得计算机的研制周期变得很长,同时也增加了设计失误的可能性,而且由于复杂指令需进行复杂的操作,有时还可能降低系统的执行速度。通过对传统的 CISC 指令系统进行测试表明,各种指令的使用频度相差很悬殊。最常使用的是一些比较简单的指令,这类指令仅占指令总数的 20%,但在各种程序中出现的频度却占 80%,其余大多数指令是功能复杂的指令,这类指令占指令总数的 80%,但其使用频度仅占 20%。因此,人们把这种情况称为“20%~80%律”。从这一事实出发,人们开始了对指令系统合理性的研究,于是基于精简指令系统的精简指令系统计算机(Reduced Instruction Set Computer,RISC)随之诞生。

RISC 的中心思想是要求指令系统简化,尽量使用寄存器-寄存器操作指令,除去访存指令(LOAD 和 STORE)外其他指令的操作均在单周期内完成,指令格式力求一致,寻址方式尽可能减少,并提高编译的效率,最终达到加快机器处理速度的目的。

3.5.3 VLIW 和 EPIC

1. VLIW 和 EPIC 概念

VLIW 是英文“Very Long Instruction Word”的缩写,中文含义是“超长指令字”,即一

种非常长的指令组合，它把许多条指令连在一起，增加了运算的速度。在这种指令系统中，编译器把许多简单、独立的指令组合到一条指令字中。当这些指令字从主存中取出放到处理器中时，它们被容易地分解成几条简单的指令，这些简单的指令被分派到一些独立的执行单元去执行。

EPIC 是英文“Explicit Parallel Instruction Code”的缩写，中文含义是“显式并行指令代码”。EPIC 是从 VLIW 中衍生出来的，通过将多条指令放入一个指令字，有效地提高了 CPU 各个计算功能部件的利用效率，提高了程序的性能。

VLIW 和 EPIC 处理器的指令集与传统处理器的指令集有极大的区别。

2. Intel IA-64 结构

虽然 80x86 指令集功勋卓著，但日显疲态也是人所共知的事实。随着时间的推移，IA-32 结构的局限性越来越明显了。作为一种 CISC 架构，变长指令结构、有无数种不同的指令格式，使它难于在执行中进行快速译码；同时，为了能够使用 RISC 架构上非常普遍的流水线和分支预测等技术，Intel 公司被迫增加了很多复杂的设计。因此，Intel 公司决定抛弃 IA-32 结构，转向全新的指令系统，20 世纪末，由 Intel 公司和 HP 公司联合推出了彻底突破 IA-32 结构的 IA-64 结构，最大限度地开发了指令级并行操作。

Intel 公司反对将 IA-64 结构划归到 RISC 或 CISC 的类别中，因为他们认为这是 EPIC 架构，是一种基于超长指令字的设计，它合并了 RISC 和 VLIW 技术方面的优势。最早采用这种技术的处理器是 Itanium，后来又有了 Itanium 2。

Itanium 有 128 个 64 位的整数寄存器、128 个 82 位的浮点寄存器、64 个 1 位的判定寄存器和 8 个 64 位的分支寄存器。Itanium 在硬件上与 IA-32 指令集兼容，通过翻译软件与 HP 公司的 PA-RISC 指令集兼容。

3. 128 位指令束

IA-64 结构将 3 条指令拼接成 128 位的“指令束”，以加快处理速度。每个指令束里包含了 3 个 41 位的指令和 1 个 5 位的模板，如图 3-17 所示。这个 5 位的模板包含了不同指令间的并行信息，编译器将使用模板告诉 CPU，哪些指令可以并行执行。模板也包含了指令束的结束位，用以告诉 CPU 这个指令束是否结束，是否需准备捆绑下两个或更多的指令束。

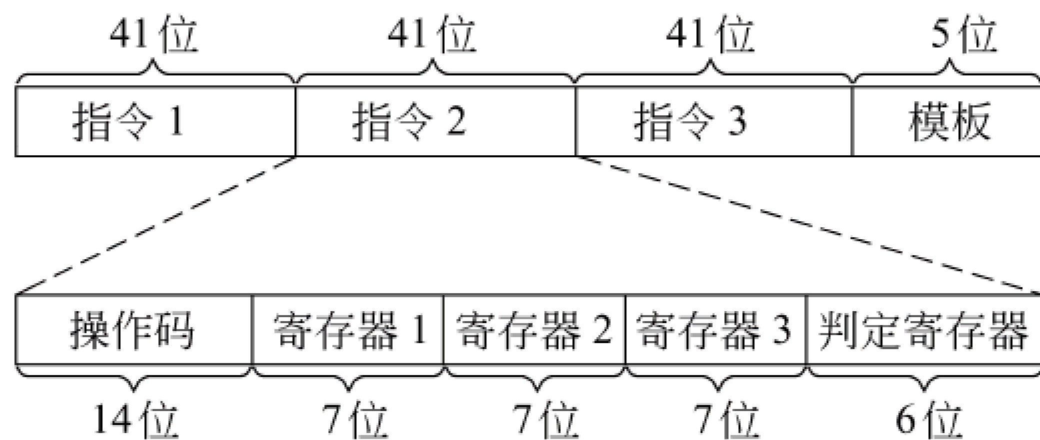


图 3-17　IA-64 结构的指令束格式

指令束中的每条指令的长度是固定的，均为 41 位，由指令操作码字段、判定寄存器字段和 3 个寄存器字段(其中 2 个为源寄存器，1 个为目的寄存器)组成，指令只对寄存器操作。

一个指令束中的3条指令之间一定是没有依赖关系的,由编译程序将3条指令拼接成指令束。假设编译程序发现了16条没有相互依赖关系的指令,便可以把它们拼接成6个不同的指令束,前5束里每束3条指令,剩下的一条指令放在第6束里,然后在模板里做上相应的标记。

指令束的128位被CPU一次装载并检测,依靠指令的模板,3条指令能被不同的执行单元同时执行。任意数目的指令束能安排在指令组里,一个指令组是一个彼此可以并行执行并且不发生冲突的指令流。

习　题

3-1　指令长度和机器字长有什么关系?半字长指令、单字长指令、双字长指令分别表示什么意思?

3-2　零地址指令的操作数来自哪里?一地址指令中,另一个操作数的地址通常可采用什么寻址方式获得?各举一例说明。

3-3　某计算机为定长指令字结构,指令长度为16位,每个操作数的地址码长为6位,指令分为无操作数、单操作数和双操作数3类。若双操作数指令已有 K 种,无操作数指令已有 L 种,问单操作数指令最多可能有多少种?上述3类指令各自允许的最大指令条数是多少?

3-4　设某计算机为定长指令字结构,指令长度为12位,每个地址码占3位,试提出一种分配方案,使该指令系统包含:4条三地址指令,8条二地址指令,180条单地址指令。

3-5　指令格式同上题,能否构成:三地址指令4条,单地址指令255条,零地址指令64条?为什么?

3-6　指令中地址码的位数与直接访问的主存容量和最小寻址单位有什么关系?

3-7　试比较间接寻址和寄存器间址。

3-8　试比较基址寻址和变址寻址。

3-9　某计算机字长为16位,主存容量为64K字,采用单字长单地址指令,共有50条指令。假设有直接寻址、间接寻址、变址寻址和相对寻址4种寻址方式,试设计其指令格式。

3-10　某计算机字长为16位,主存容量为64K字,指令格式为单字长单地址,共有64条指令。试说明:

(1) 若只采用直接寻址方式,那么指令能访问多少主存单元?

(2) 为扩充指令的寻址范围,可采用直接/间接寻址方式,若只增加一位直接/间接标志,那么指令可寻址范围为多少?指令直接寻址的范围为多少?

(3) 采用页面寻址方式,若只增加一位Z/C(零页/现行页)标志,那么指令寻址范围为多少?指令直接寻址范围为多少?

(4) 将(2)、(3)两种方式结合,指令的寻址范围为多少?指令直接寻址范围为多少?

3-11　设某计算机字长为32位,CPU有32个32位的通用寄存器,设计一个能容纳64种操作的单字长指令系统。

(1) 如果是存储器间接寻址方式的寄存器-存储器型指令,那么直接寻址的最大主存空间是多少?

(2) 如果采用通用寄存器作为基址寄存器,那么直接寻址的最大主存空间又是多少?

3-12　已知某小型机字长为16位,其双操作数指令的格式如下:

0　　　　5	6　　7	8　　　　15
OP	R	A

其中,OP为操作码,R为通用寄存器地址,试说明下列各种情况下能访问的最大主存区域有多少机器字。

(1) A 为立即数。

(2) A 为直接主存单元地址。

(3) A 为间接地址(非多重间址)。

(4) A 为变址寻址的形式地址,假定变址寄存器为 R_1(字长为 16 位)。

3-13 计算下列 4 条指令的有效地址(指令长度为 16 位)。

(1) 000000Q (2) 100000Q (3) 170710Q (4) 012305Q

假设:上述 4 条指令均用八进制书写,指令的最左边是一位间址指示位@(@=0,直接寻址;@=1,间接寻址),且具有多重间访功能;指令的最右边两位为形式地址;主存容量为 2^{15} 个单元,表 3-4 为有关主存单元的内容(八进制)。

表 3-4 习题 3-13 的主存单元内容

地址	内容	地址	内容
00000	100002	00005	100001
00001	046710	00006	063215
00002	054304	00007	077710
00003	100000	00010	100005
00004	102543		

3-14 假定某计算机的指令格式如下:

11	10 9	8	7	6	5 … 0
@	OP	I_1	I_2	Z/C	A

其中,

bit11=1:间接寻址。

bit8=1:变址寄存器 I_1 寻址。

bit7=1:变址寄存器 I_2 寻址。

bit6(零页/现行页寻址):Z/C=0,表示 0 页面;Z/C=1,表示现行页面,即指令所在页面。

若主存容量为 2^{12} 个存储单元,分为 2^6 个页面,每个页面有 2^6 个字。

设有关寄存器的内容为:

(PC)=0340Q (I_1)=1111Q (I_2)=0256Q

试计算下列指令的有效地址。

(1) 1046Q (2) 2433Q (3) 3215Q (4) 1111Q

3-15 假定指令格式如下:

15 12	11	10	9	8	7 … 0
OP	I_1	I_2	Z/C	D/I	A

其中,D/I 为直接/间接寻址标志,D/I=0 表示直接寻址,D/I=1 表示间接寻址。其余标志位同习题 3-14 的说明。

若主存容量为 2^{16} 个存储单元,分为 2^8 个页面,每个页面有 2^8 个字。

设有关寄存器的内容为:

(I_1)=002543Q (I_2)=063215Q (PC)=004350Q

试计算下列指令的有效地址。

(1) 152301Q (2) 074013Q (3) 161123Q (4) 140011Q

3-16　设某计算机有变址寻址、间接寻址和相对寻址等寻址方式,当前指令的地址码部分为001AH,正在执行的指令所在地址为1F05H,变址寄存器中的内容为23A0H。

(1) 当执行取数指令时,如为变址寻址方式,则取出的数是多少?

(2) 如为间接寻址,则取出的数是多少?

(3) 当执行转移指令时,转移地址是多少?

已知主存部分地址及相应内容见表3-5。

表3-5　习题3-16的主存部分地址及相应内容

地址	内容	地址	内容
001AH	23A0H	23A0H	2600H
1F05H	241AH	23BAH	1748H
1F1FH	2500H		

3-17　请举例说明,哪几种寻址方式除去取指令以外不访问存储器?哪几种寻址方式除去取指令以外只需访问一次存储器?完成什么样的指令(包括取指令在内)共需访问4次存储器?

3-18　设相对寻址的转移指令占两个字节,第一个字节是操作码,第二个字节是相对位移量,用补码表示。假设当前转移指令第一字节所在的地址为2000H,且CPU每取一个字节便自动地完成(PC)+1→PC的操作。试问:当执行JMP * +8和JMP * -9指令(*为相对寻址特征)时,转移指令第二字节的内容各为多少?转移的目的地址各是什么?

3-19　在某堆栈计算机中,用一地址指令PUSH、POP及零地址指令ADD、MPY写出计算下式

$$Z=(A\times(B+C+D)\times E+F\times F)\times(B+C+D)$$

的程序。

3-20　如果在上题中增加一条DUP指令,该指令的功能是将栈顶内容复制一次。问:上述程序如何简化?

3-21　什么叫主程序和子程序?调用子程序时还可采用哪几种方法保存返回地址?画图说明调用子程序的过程。

3-22　在某些计算机中,调用子程序的方法是这样实现的:转子指令将返回地址存入子程序的第一个字单元,然后从第二个字单元开始执行子程序,回答下列问题:

(1) 为这种方法设计一条从子程序转到主程序的返回指令。

(2) 在这种情况下,如何在主、子程序间进行参数的传递?

(3) 上述方法是否可用于子程序的嵌套?

(4) 上述方法是否可用于子程序的递归(即某个子程序自己调用自己)?

(5) 如果改用堆栈方法,是否可实现(4)所提出的问题?

第 4 章 数值的机器运算

运算器是计算机进行算术运算和逻辑运算的主要部件，运算器的逻辑结构取决于机器的指令系统、数据表示方法和运算方法等。本章主要讨论数值数据在计算机中实现算术运算和逻辑运算的方法，以及运算部件的基本结构和工作原理。

4.1 基本算术运算的实现

计算机中最基本的算术运算是加法运算，加、减、乘、除运算最终都可以归结为加法运算。所以，在此讨论最基本的运算部件——加法器，以及并行加法器的进位问题。

4.1.1 加法器

加法器是由全加器再配以其他必要的逻辑电路组成的。

1. 全加器

全加器（FA）是最基本的加法单元，它有 3 个输入量：操作数 A_i 和 B_i，低位传来的进位 C_{i-1}。两个输出量：本位和 S_i，向高位的进位 C_i。全加器的逻辑框图如图 4-1 所示，其真值表如表 4-1 所示。

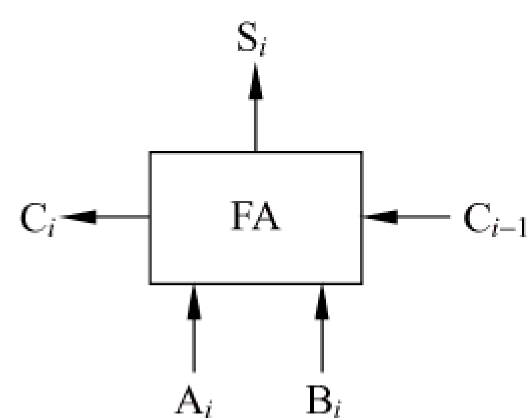


图 4-1　全加器

表 4-1　全加器真值表

A_i	B_i	C_{i-1}	S_i	C_i
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

根据真值表，可得到全加器的逻辑表达式：

$$S_i = A_i \oplus B_i \oplus C_{i-1}$$
$$C_i = A_iB_i + (A_i \oplus B_i)C_{i-1}$$

2. 串行加法器与并行加法器

加法器有串行和并行之分。在串行加法器中,只有一个全加器,数据逐位串行送入加法器进行运算;并行加法器则由多个全加器组成,其位数的多少取决于机器的字长,数据的各位同时运算。

串行加法器如图 4-2 所示。图中 FA 是全加器,A、B 是两个具有右移功能的寄存器,C 为进位触发器。由移位寄存器从低位到高位逐位串行提供操作数相加。如果操作数长 n 位,加法就要分 n 次进行,每次产生一位和,且串行地送回 A 寄存器。进位触发器用来寄存进位信号,以便参与下一次的运算。

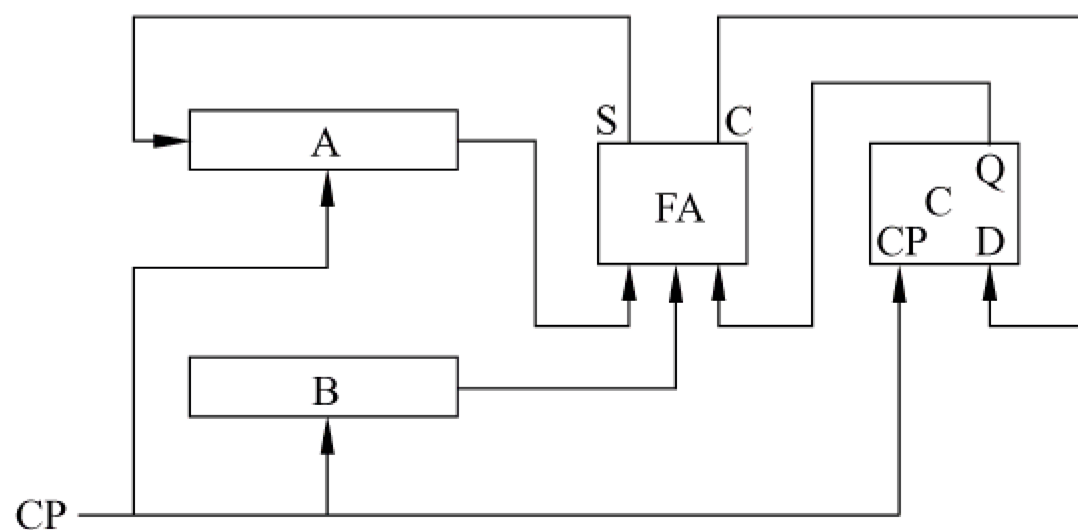


图 4-2 串行加法器

串行加法器具有器件少、成本低的优点,但运算速度太慢,所以除了某些低速的专用运算器外很少采用。

并行加法器可同时对数据的各位相加,但存在着一个加法的最长运算时间问题。这是因为虽然操作数的各位是同时提供的,但低位运算所产生的进位会影响高位的运算结果。例如,11…11 和 00…01 相加,最低位产生的进位将逐位影响至最高位,因此,并行加法器的最长运算时间主要是由进位信号的传递时间决定的,而每个全加器本身的求和延迟只是次要因素。很明显,提高并行加法器速度的关键是尽量加快进位产生和传递的速度。

4.1.2 进位的产生和传递

并行加法器中的每一个全加器都有一个从低位送来的进位输入和一个传送给高位的进位输出。通常将传递进位信号的逻辑线路连接起来构成的进位网络称为进位链。每一位的进位表达式为:

$$C_i = A_iB_i + (A_i \oplus B_i)C_{i-1}$$

其中,“A_iB_i”取决于本位参加运算的两个数,而与低位进位无关,因此称 A_iB_i 为进位产生函数(本次进位产生),用 G_i 表示,其含义是:若本位的两个输入均为 1,必然要向高位产生进位。“$(A_i \oplus B_i)C_{i-1}$”不但与本位的两个数有关,还依赖于低位送来的进位,因此称 $A_i \oplus B_i$ 为进位传递函数(低位进位传递),用 P_i 表示,其含义是:当两个输入中有一个为 1,低位传来的进位 C_{i-1} 将向更高位传送,所以进位表达式又可以写成:

$$C_i = G_i + P_iC_{i-1}$$

把 n 个全加器串接起来，就可进行两个 n 位数的相加。这种加法器称为串行进位的并行加法器，如图 4-3 所示。串行进位又称行波进位，每一级进位直接依赖于前一级的进位，即进位信号是逐级形成的。

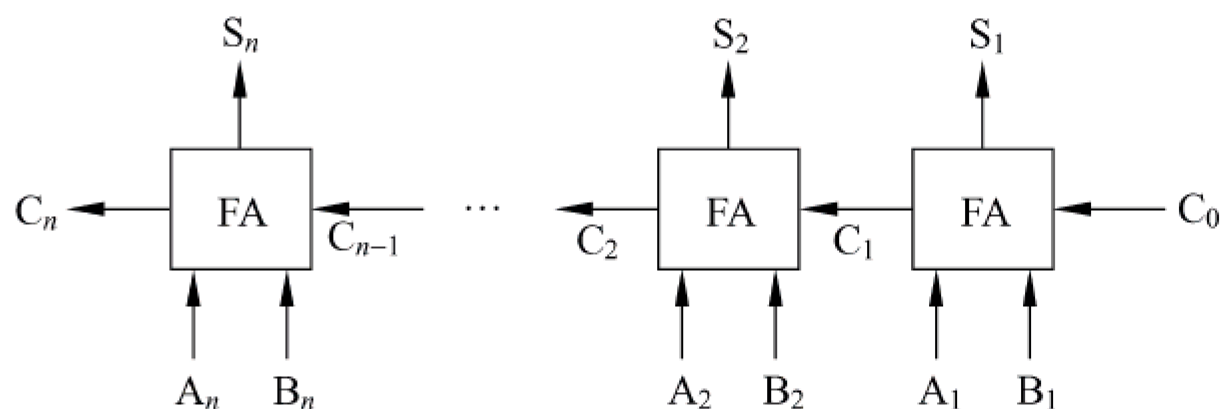


图 4-3　串行进位的并行加法器

其中：

$$C_1 = G_1 + P_1C_0$$
$$C_2 = G_2 + P_2C_1$$
$$\vdots$$
$$C_n = G_n + P_nC_{n-1}$$

串行进位的并行加法器的总延迟时间与字长成正比，字长越长，则总延迟时间越长。假设将一级“与门”“或门”的延迟时间定为 ty，从上述公式中可看出，每一级全加器的进位延迟时间为 2ty。在字长为 n 位的情况下，若不考虑 G_i、P_i 的形成时间，从 $C_0 \rightarrow C_n$ 的最长延迟时间为 $2n$ty（设 C_0 为加法器最低位的进位输入，C_n 为加法器最高位的进位输出）。

4.1.3　并行加法器的快速进位

显然，串行进位方式的进位延迟时间太长了，要提高加法运算的速度，就要尽可能地减少进位延迟时间，也就是要改进进位方式。

1. 并行进位方式

并行进位又称先行进位、同时进位，其特点是各级进位信号同时形成。

$$C_1 = G_1 + P_1C_0$$
$$C_2 = G_2 + P_2C_1 = G_2 + P_2G_1 + P_2P_1C_0$$
$$C_3 = G_3 + P_3C_2 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1C_0$$
$$C_4 = G_4 + P_4C_3 = G_4 + P_4G_3 + P_4P_3G_2 + P_4P_3P_2G_1 + P_4P_3P_2P_1C_0$$
$$\vdots$$

上述各式中所有的进位输出仅由 G_i、P_i 及最低进位输入 C_0 决定，而不依赖于其低位的进位输入 C_{i-1}，因此各级进位输出可以同时产生。这种进位方式是快速的，若不考虑 G_i、P_i 的形成时间，从 $C_0 \rightarrow C_n$ 的最长延迟时间仅为 2ty，而与字长无关。但是，随着加法器位数的增加，C_i 的逻辑表达式会变得越来越长，输入变量会越来越多，这会使电路结构变得很复杂，所以，完全采用并行进位是不现实的。

2. 分组并行进位方式

实际上，通常采用分组并行进位方式。这种进位方式是把 n 位字长分为若干小组，在组内各位之间实行并行快速进位，在组间既可以采用串行进位方式，也可以采用并行快速进位

方式,因此有两种情况。

(1) 单级先行进位方式(组内并行、组间串行)

以16位加法器为例,可分为4组,每组4位。第一小组组内的进位逻辑函数 C_1、C_2、C_3、C_4 的表达式与前述相同,$C_1 \sim C_4$ 信号是同时产生的,实现上述进位逻辑函数的电路称为4位先行进位电路(Carry Look Ahead,CLA),其延迟时间是2ty。

利用这种4位的CLA电路以及进位产生/传递电路和求和电路可以构成4位的CLA加法器。用4个这样的CLA加法器,很容易构成16位的单级先行进位加法器,如图4-4所示。

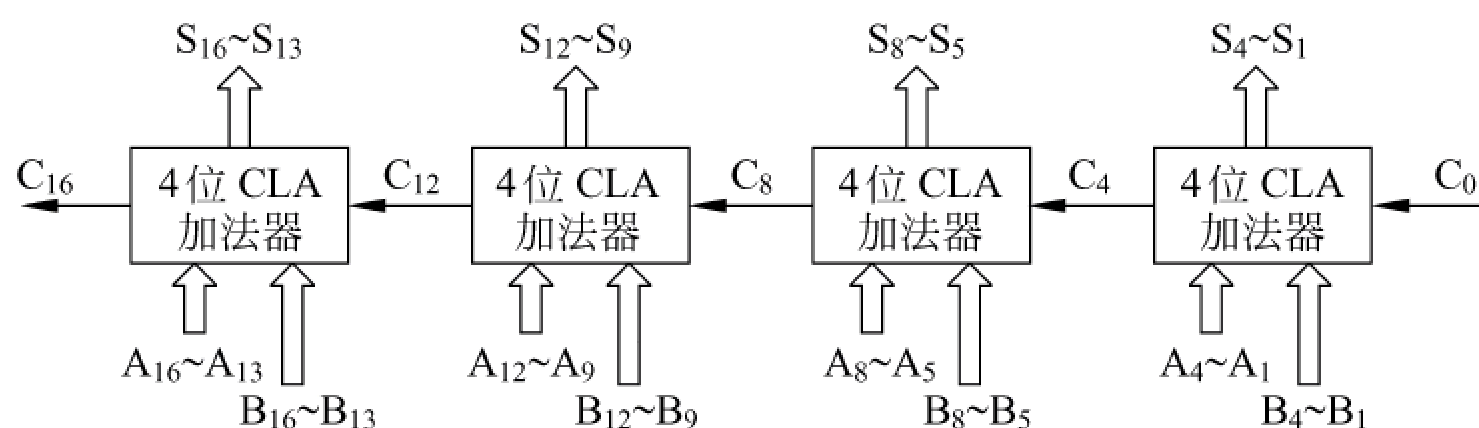


图4-4　16位单级先行进位加法器

若不考虑 G_i、P_i 的形成时间,从 $C_0 \rightarrow C_n$ 的最长延迟时间为 $2m$ty,其中 m 为分组的组数。16位单级先行进位加法器,从 $C_1 \sim C_{16}$ 的最长延迟时间为 $4 \times 2\text{ty} = 8\text{ty}$,图4-5是这种加法器的进位时间图。

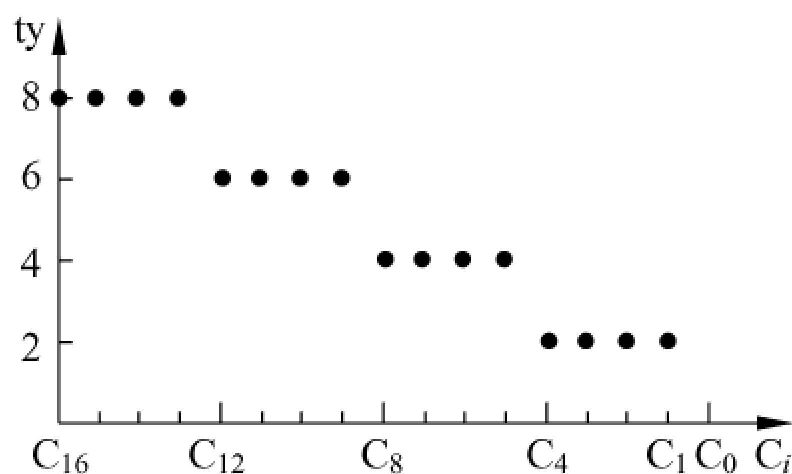


图4-5　16位单级先行进位时间图

(2) 多级先行进位方式(组内并行、组间并行)

在单级先行进位电路中,进位的延迟时间是和组数成正比的,组数越多,则进位延迟时间越长,因此当加法器的字长较长($n \geqslant 16$)时,为了加快进位传递时间,就有必要采用多级先行进位方式。

下面仍以字长为16位的加法器作为例子,分析两级先行进位加法器的设计方法。第一小组的进位输出 C_4 可以写为:

$$C_4 = G_4 + P_4G_3 + P_4P_3G_2 + P_4P_3P_2G_1 + P_4P_3P_2P_1C_0 = G_1^* + P_1^*C_0$$

其中,$G_1^* = G_4 + P_4G_3 + P_4P_3G_2 + P_4P_3P_2G_1$,$P_1^* = P_4P_3P_2P_1$。

G_i^* 称为组进位产生函数,P_i^* 称为组进位传递函数,这两个辅助函数只与 P_i、G_i 有关。依此类推,可以得到:

$$C_8 = G_2^* + P_2^*C_4 = G_2^* + P_2^*G_1^* + P_2^*P_1^*C_0$$

$$C_{12} = G_3^* + P_3^*G_2^* + P_3^*P_2^*G_1^* + P_3^*P_2^*P_1^*C_0$$

$$C_{16} = G_4^* + P_4^*G_3^* + P_4^*P_3^*G_2^* + P_4^*P_3^*P_2^*G_1^* + P_4^*P_3^*P_2^*P_1^*C_0$$

为了要产生组进位函数,需要对原来的CLA电路进行修改:

第一小组内产生 G_1^*、P_1^*、C_3、C_2、C_1,不产生 C_4;

第二小组内产生 G_2^*、P_2^*、C_7、C_6、C_5,不产生 C_8;

第三小组内产生 G_3^*、P_3^*、C_{11}、C_{10}、C_9,不产生 C_{12};

第四小组内产生 G_4^*、P_4^*、C_{15}、C_{14}、C_{13},不产生 C_{16}。

这种电路称为成组先行进位电路(Block Carry Look Ahead,BCLA),其延迟时间是2ty。利用这种4位的BCLA电路以及进位产生与传递电路和求和电路可以构成4位的BCLA加法器。16位的两级先行进位加法器可由4个BCLA加法器和1个CLA电路组成,如图4-6所示。

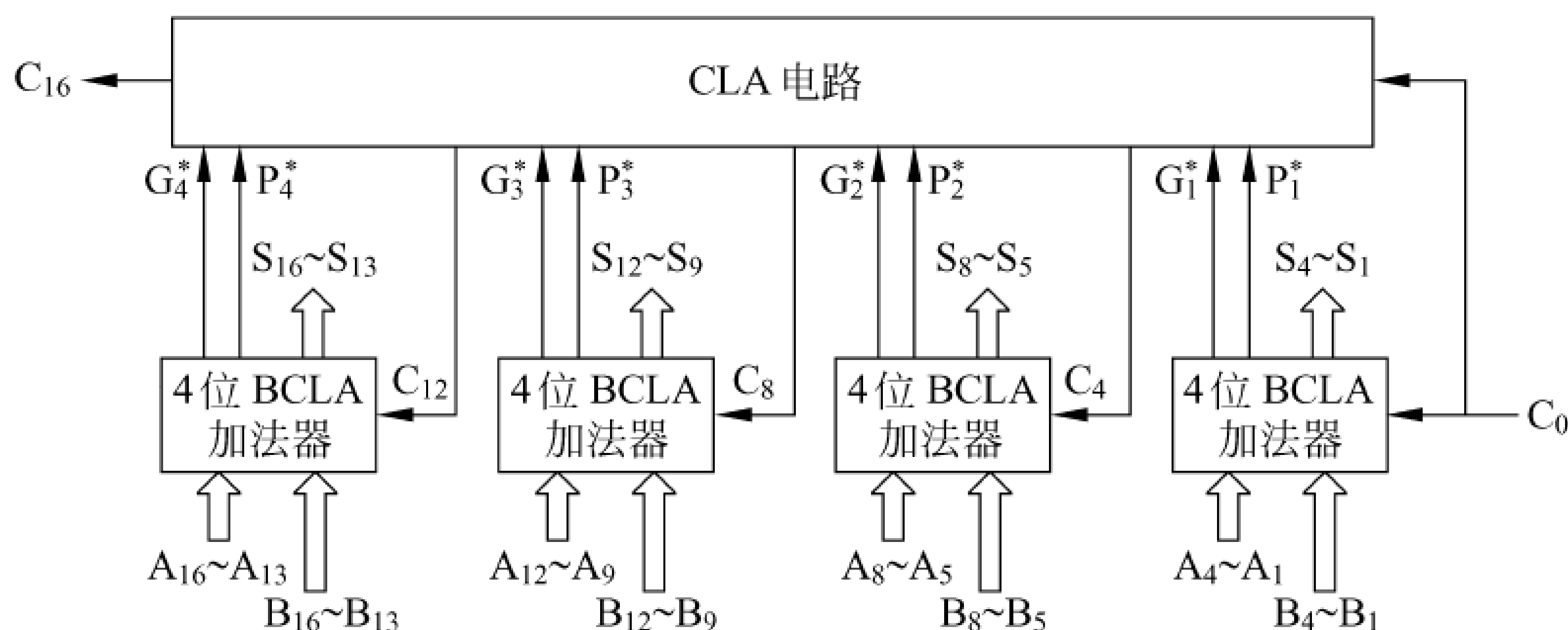


图 4-6　16 位两级先行进位加法器

由图4-6可见,若不考虑G_i、P_i的形成时间,C_0经过2ty产生第一小组的C_1、C_2、C_3及所有组进位产生函数G_i^*和组进位传递函数P_i^*;再经过2ty,由CLA电路产生C_4、C_8、C_{12}、C_{16};再经过2ty后,才能产生第二、三、四小组内的$C_5\sim C_7$、$C_9\sim C_{11}$、$C_{13}\sim C_{15}$。它的进位时间图如图4-7所示,此时加法器的最长进位延迟时间是6ty。

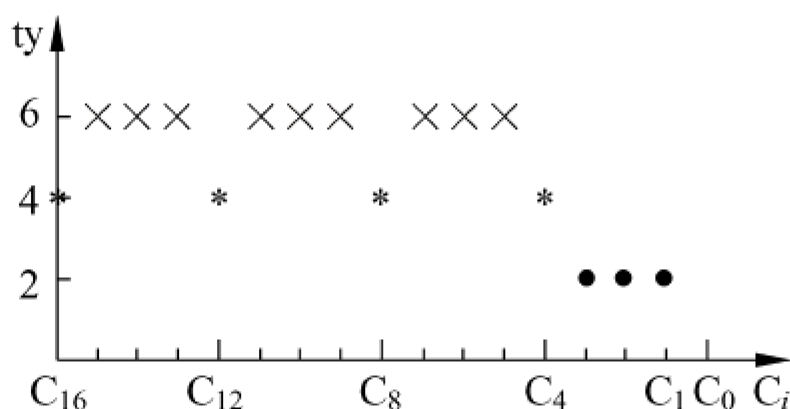


图 4-7　16 位两级先行进位时间图

用同样的方法可以扩展到多于两级的先行进位加法器,如用三级先行进位结构设计64位加法器。这种加法器的字长对加法时间影响甚小,但造价较高。

注意:从图4-4和图4-6中可以看出,4位CLA加法器和4位BCLA加法器的区别仅在于其中的进位逻辑电路上,前者产生进位输出信号$C_4\sim C_1$,后者产生进位输出信号$C_3\sim C_1$及组进位产生和传递信号G_1^*、P_1^*。

4.2　定点加减运算

定点数的加减运算包括原码、补码和反码3种带符号数的加减运算,其中补码加减运算实现起来最方便。

4.2.1　原码加减运算

对两个数进行加减运算时,计算机的实际操作是加还是减,不仅取决于指令的操作码,还取决于两个操作数的符号。例如,加法运算时可能要做减法操作(两数异号);而减法运算时又可能做加法操作(两数异号)。当原码加减运算时,符号位并不参加运算,只有两数的绝对值参加运算。首先要判断参加运算的两个操作数的符号,再根据要求决定进行相加还是相减操作,最后还要根据两个操作数绝对值的大小决定结果的符号,整个运算过程比较复杂。

在大多数计算机中,通常只设置加法器而不设置减法器,减法运算将转换为加法运算来实现。原码运算时,用$|X|+[|Y|]_{变补}$来代替$|X|-|Y|$。

原码加减运算规则如下:

① 参加运算的操作数取其绝对值。

② 若做加法运算,则两数直接相加;若做减法运算,则将减数先变一次补,再进行加法操作。

③ 运算之后,可能有两种情况:

- 有进位,结果为正,即得到正确的结果。
- 无进位,结果为负,则应再变一次补,才能得到正确的结果。

④ 结果加上符号位。

通常,把运算之前的变补称为前变补,运算之后的变补称为后变补。

注意:变补是指所有的二进制数各位变反后最低位加1。

4.2.2 补码加减运算

补码加减运算要比原码加减运算简单得多。

1. 补码加法

两个补码表示的数相加,符号位参加运算,且两数和的补码等于两数补码之和,即:

$$[X+Y]_补=[X]_补+[Y]_补$$

根据补码定义,分4种情况可以证明该式的正确性。

(1) $X>0,Y>0$

因为$X>0,Y>0$,则$(X+Y)>0$。

由补码定义: $[X]_补=X,\quad [Y]_补=Y$

所以 $[X]_补+[Y]_补=X+Y=[X+Y]_补$

(2) $X<0,Y<0$

因为$X<0,Y<0$,则$(X+Y)<0$。

由补码定义: $[X]_补=M+X,\quad [Y]_补=M+Y \pmod M$

所以 $[X]_补+[Y]_补=M+X+M+Y=M+(M+X+Y)=M+[X+Y]_补$

$=[X+Y]_补 \pmod M$

(3) $X>0,Y<0$

由补码定义: $[X]_补=X,\quad [Y]_补=M+Y \pmod M$

$[X]_补+[Y]_补=X+M+Y=M+X+Y$

有以下两种情况:

① 当$(X+Y)\geqslant 0$时,M被丢掉,因此:

$$[X]_补+[Y]_补=M+X+Y=[X+Y]_补 \pmod M$$

② 当$(X+Y)<0$时,由补码定义有:

$$[X]_补+[Y]_补=M+X+Y=[X+Y]_补 \pmod M$$

(4) $X<0,Y>0$

情况与(3)类似,只要将X、Y位置对调即可证明。

2. 补码减法

也可以借用加法器来实现减法运算，根据补码加法公式可以推出：

$$[X-Y]_{补}=[X+(-Y)]_{补}=[X]_{补}+[-Y]_{补}$$

从补码减法公式可以看出，只要求得$[-Y]_{补}$，就可以变减法为加法。下面以定点整数为例证明由$[Y]_{补}$求$[-Y]_{补}$的方法，此法很容易推广到定点小数。

设$[Y]_{补}=Y_s,Y_1Y_2\cdots Y_n$，为便于观察将符号位$Y_s$与数值位之间用逗号分开。现有下列两种情况。

(1) $0\leqslant Y<2^n$

已知：$$[Y]_{补}=[Y]_{原}=0,Y_1Y_2\cdots Y_n$$

故 $$[-Y]_{原}=1,Y_1Y_2\cdots Y_n$$

Y 为正数，$-Y$ 则为负数，根据原码求补码的方法，可以得到：

$$[-Y]_{补}=1,\overline{Y_1}\ \overline{Y_2}\cdots\overline{Y_n}+1$$

(2) $-2^n<Y<0$

已知：$$[Y]_{补}=1,Y_1Y_2\cdots Y_n$$

根据补码求原码的方法， $$[Y]_{原}=1,\overline{Y_1}\ \overline{Y_2}\cdots\overline{Y_n}+1$$

由于 Y 是负数，则$-Y$ 为正数，有：

$$[-Y]_{补}=[-Y]_{原}=0,\overline{Y_1}\ \overline{Y_2}\cdots\overline{Y_n}+1$$

综合以上两种情况，不管 Y 的真值为正或为负，已知$[Y]_{补}$求$[-Y]_{补}$的方法是：将$[Y]_{补}$连同符号位一起求反，末尾加 1（在定点小数中这个"1"实际上是 2^{-n}）。将$[-Y]_{补}$称为$[Y]_{补}$的机器负数，由$[Y]_{补}$求$[-Y]_{补}$的过程称为对$[Y]_{补}$变补(求补)，表示为：

$$[-Y]_{补}=[[Y]_{补}]_{变补}$$

注意：应将"某数的补码表示"与"变补"这两个概念区分开来。一个负数由原码表示转换成补码表示时，符号位是不变的，仅对数值位各位变反，末位加"1"。而变补则不论这个数的真值是正是负，一律连同符号位一起变反(所有的二进制位一起变反)，末位加"1"。$[Y]_{补}$表示的真值如果是正数，则变补后$[-Y]_{补}$所表示的真值变为负数，反之亦然。

例 4-1 设 $Y=-0.0110$。则有：

$$[Y]_{原}=1.0110,\quad [Y]_{补}=1.1010,\quad [-Y]_{补}=0.0110$$

例 4-2 设 $Y=0.0110$。则有：

$$[Y]_{原}=0.0110,\quad [Y]_{补}=0.0110,\quad [-Y]_{补}=1.1010$$

3. 补码加减运算规则

补码加减运算规则如下：

① 参加运算的两个操作数均用补码表示。

② 符号位作为数的一部分参加运算。

③ 若做加法运算，则两数直接相加；若做减法运算，则将被减数与减数的机器负数相加。

④ 运算结果仍用补码表示。

例 4-3 设 $A=0.1011$，$B=-0.1110$，求 $A+B$。

其中： $[A]_补=0.1011$， $[B]_补=1.0010$

$$
\begin{array}{rl}
 & 0.1011 \quad [A]_补 \\
+ & 1.0010 \quad [B]_补 \\
\hline
 & 1.1101 \quad [A+B]_补
\end{array}
$$

故 $$[A+B]_补=1.1101$$
$$A+B=-0.0011$$

例 4-4 设 $A=0.1011$，$B=-0.0010$，求 $A-B$。

其中： $[A]_补=0.1011$， $[B]_补=1.1110$， $[-B]_补=0.0010$

$$
\begin{array}{rl}
 & 0.1011 \quad [A]_补 \\
+ & 0.0010 \quad [-B]_补 \\
\hline
 & 0.1101 \quad [A-B]_补
\end{array}
$$

故 $$[A-B]_补=0.1101$$
$$A-B=0.1101$$

4. 符号扩展

在计算机算术运算中，有时必须将采用给定位数表示的数转换成具有更多位数的某种表示形式。例如，某个程序需要将一个8位数与另外一个32位数相加。要想得到正确的结果，在将8位数与32位数相加之前，必须将8位数转换成32位数形式，这个过程称为"符号扩展"。

对于正数的符号扩展非常简单，原有形式的符号位移动到新形式的符号位上，新表示形式的所有附加位都用"0"进行填充。

对于负数的符号扩展方法则根据机器数的不同而不同。原码表示负数的符号扩展方法与正数相同，只不过此时符号位为"1"而已。补码表示负数的扩展方法是：原有形式的符号位移动到新形式的符号位上，新表示形式的所有附加位都用"1"进行填充。

综上所述，实际上补码的符号扩展非常简单，所有附加位均用符号位填充，即正数用"0"填充，负数用"1"填充。

4.2.3 补码的溢出判断与检测方法

1. 溢出的产生

在补码加减运算中，有时会遇到这样的情况：两个正数相加，而结果的符号位却为1(结果为负)；两个负数相加，而结果的符号位却为0(结果为正)，现以字长为5位的定点整数的加法运算举例如下。

例 4-5 设 $X=1011B=11D$，$Y=111B=7D$。则有：

$$[X]_补=0,1011,\quad [Y]_补=0,0111$$

$$
\begin{array}{rl}
 & 0,1\,0\,1\,1 \quad [X]_补 \\
+ & 0,0\,1\,1\,1 \quad [Y]_补 \\
\hline
 & 1,0\,0\,1\,0 \quad [X+Y]_补
\end{array}
$$

故　$[X+Y]_补=1,0010$

$X+Y=-1110B=-14D$

两正数相加结果为−14D，显然是错误的。

例 4-6　设 $X=-1011B=-11D, Y=-111B=-7D$。则有：

$[X]_补=1,0101$，　$[Y]_补=1,1001$

$$\begin{array}{rl} 1,0\,1\,0\,1 & [X]_补 \\ +\ 1,1\,0\,0\,1 & [Y]_补 \\ \hline 0,1\,1\,1\,0 & [X+Y]_补 \end{array}$$

故　$[X+Y]_补=0,1110$

$X+Y=1110B=14D$

两负数相加结果为14D，显然也是错误的。

为什么会发生这种错误呢？原因在于两数相加之和的数值已超过了机器允许的表示范围。字长为 $n+1$ 位的定点整数(其中一位为符号位)，采用补码表示，当运算结果大于 2^n-1 或小于 -2^n 时，就产生溢出。

设参加运算的两数为 X 和 Y，做加法运算。

- 若 X 和 Y 异号，实际上是做两数相减，所以不会溢出。
- 若 X 和 Y 同号，运算结果为正且大于所能表示的最大正数或运算结果为负且小于所能表示的最小负数(绝对值最大的负数)时，产生溢出。将两正数相加产生的溢出称为正溢；反之，两负数相加产生的溢出称为负溢。

2. 溢出检测方法

假设，被操作数为：　$[X]_补=X_s,X_1X_2\cdots X_n$

操作数为：　$[Y]_补=Y_s,Y_1Y_2\cdots Y_n$

其和(差)为：　$[S]_补=S_s,S_1S_2\cdots S_n$

(1) 采用一个符号位

从前述两个例子还可以看出，采用一个符号位检测溢出时，当 $X_s=Y_s=0, S_s=1$ 时，产生正溢；当 $X_s=Y_s=1, S_s=0$ 时，产生负溢。

溢出判断条件为：

$$溢出=\overline{X}_s\overline{Y}_sS_s+X_sY_s\overline{S}_s$$

(2) 采用进位位

两数运算时，产生的进位为：

$$C_s,C_1C_2\cdots C_n$$

其中，C_s 为符号位产生的进位，C_1 为最高数值位产生的进位。

从前述两个例子还可以看出，两正数相加，当最高有效位产生进位($C_1=1$)而符号位不产生进位($C_s=0$)时，发生正溢；两负数相加，当最高有效位不产生进位($C_1=0$)而符号位产生进位($C_s=1$)时，发生负溢。故溢出条件为：

$$溢出=\overline{C}_sC_1+C_s\overline{C}_1=C_s\oplus C_1$$

(3) 采用变形补码(双符号位补码)

一个符号位只能表示正、负两种情况,当产生溢出时,符号位的含义就会发生混乱。如果将符号位扩充为两位(S_{s1}和S_{s2}),其所能表示的信息量将随之扩大,既能检测出是否溢出,又能指出结果的符号。在双符号位的情况下,把左边的符号位S_{s1}称为真符,因为它代表了该数真正的符号,两个符号位都作为数的一部分参加运算。这种编码又称为变形补码。

双符号位的含义如下:

$S_{s1}S_{s2}=00$　　结果为正数,无溢出

$S_{s1}S_{s2}=01$　　结果正溢

$S_{s1}S_{s2}=10$　　结果负溢

$S_{s1}S_{s2}=11$　　结果为负数,无溢出

当两位符号位的值不一致时,表明产生溢出,溢出条件为:

$$溢出=S_{s1}\oplus S_{s2}$$

如果前述的例子采用了双符号位,则有:

11+7=18(结果大于最大正数15)

$$\begin{array}{r} 00,1011 \\ +\ 00,0111 \\ \hline 01,0010 \end{array} \quad 正溢$$

−11+(−7)=−18(结果小于绝对值最大的负数−16)

$$\begin{array}{r} 11,0101 \\ +\ 11,1001 \\ \hline 10,1110 \end{array} \quad 负溢$$

双符号位实质上是扩大了模,对于定点小数来说,模等于4;对于字长为$n+2$位的定点整数来说,模等于2^{n+2}。

定点小数的变形补码定义为:

$$[X]_{补}=\begin{cases} X & 0\leqslant X<1 \\ 4+X & -1\leqslant X<0 \end{cases} \quad (\text{mod } 4)$$

字长为$n+2$位的定点整数的变形补码定义为:

$$[X]_{补}=\begin{cases} X & 0\leqslant X<2^n \\ 2^{n+2}+X & -2^n\leqslant X<0 \end{cases} \quad (\text{mod } 2^{n+2})$$

为了尽可能减少代价,在采用双符号位方案时,操作数和结果在寄存器和主存中仍保持单符号位,仅在运算时再扩充为双符号位。

4.2.4 补码定点加减运算的实现

实现补码加减运算的逻辑电路如图4-8所示。

图4-8中F代表一个多位的并行加法器,其功能是:接收参加运算的两个数,进行加法运算,并在输出端给出本次运算结果。X和Y是两个寄存器,用来存放参加运算的数据,寄存器X同时还用来保存运算结果。门A、B、C分别是字级的与门和与或门,门A用来控制把寄存器X各位的输出送到加法器F的左输入端,其控制信号为X→F;门C用来控制把加法器F各位的运算结果送回寄存器X,其控制信号为F→X;门B则通过两个不同的控制信号Y→F和$\overline{\text{Y}}$→F,分别实现把寄存器Y各位的内容(即各触发器的Q端)送加法器F,或实

图 4-8　补码加减运算器

现把寄存器 Y 各位的内容取反后(即各触发器的 $\overline{Q}$ 端)送加法器 F。加法器 F 最低位还有一个进位控制信号 1→F。CP_X是寄存器 X 的打入脉冲。

若要实现补码加法,则需给出 X→F、Y→F 和 F→X 这 3 个控制信号,同时打开门 A、门 B 和门 C,把寄存器 X 和寄存器 Y 的内容送入加法器的两个输入端进行加法运算,并把结果送回,最后由打入脉冲 CP_X打入寄存器 X。

减法与加法的不同之处在于,加法使用 Y→F 控制信号,减法使用 $\overline{Y}$→F 和 1→F 控制信号,其余控制信号相同。

4.3　带符号数的移位和舍入操作

在计算机中,实现乘除运算的方案通常有以下 3 种:

① 软件实现。在低档微机中无乘除运算指令,只能用乘法和除法子程序来实现乘除运算。

② 在原有实现加减运算的运算器基础上增加一些逻辑线路,使乘除运算变换成加减和移位操作。在机器中设有乘除指令。

③ 设置专用的乘、除法器,机器中设有相应的乘除指令。

不管采用什么方案实现乘除法,基本原理是相同的。如果采用第②种方案,则必然会涉及移位操作。

4.3.1　带符号数的移位操作

在第 3 章中讨论过,算术移位时应保持数的符号位不变,而数值的大小则要发生变化。左移一位相当于该数乘以 2,而右移一位相当于该数除以 2。

1. 原码的移位规则

不论正数还是负数,在左移或右移时,符号位均不变,空出位一律以“0”补入。

负数的原码移位前后结果如下。

左移:移位前有　$1\ X_1\ X_2 \cdots\ X_{n-1}\ X_n$

　　　移位后有　$1\ X_2\ X_3 \cdots\ X_n\quad 0$

右移:移位前有　$1\ X_1\ X_2 \cdots\ X_{n-1}\ X_n$

　　　移位后有　$1\quad 0\quad X_1 \cdots\ X_{n-2}\ X_{n-1}$

2. 补码的移位规则

(1) 正数

符号位不变,不论左移或右移,空出位一律以“0”补入。

(2) 负数

符号位不变,左移后的空出位补“0”,右移后的空出位补“1”。

左移:移位前有 1 X_1 $X_2 \cdots$ X_{n-1} X_n

移位后有 1 X_2 $X_3 \cdots$ X_n 0

右移:移位前有 1 X_1 $X_2 \cdots$ X_{n-1} X_n

移位后有 1 1 $X_1 \cdots$ X_{n-2} X_{n-1}

3. 移位功能的实现

在计算机中,通常移位操作由移位寄存器来实现,但也有一些计算机不设置专门的移位寄存器,而在加法器的输出端加一个移位器。移位器是由与门和或门组成的逻辑电路(实际上是一个多路选择器),可以实现直传(不移位)、左斜一位送(左移一位)和右斜一位送(右移一位)的功能。移位器逻辑电路如图 4-9 所示,分别用 2F→L、F→L 和 F/2→L 这 3 个不同的控制信号选择左移、直传和右移操作。

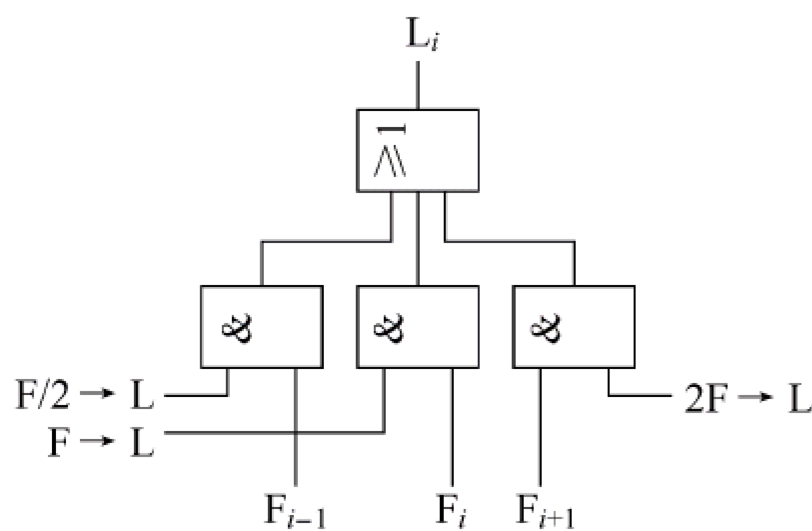


图 4-9 移位器

假设 F_0 为加法器的最高位,F_n 为加法器的最低位。左移相当于乘以 2,用 2F→L 信号控制,将 F_{i+1} 送到 L_i;右移相当于除以 2,用F/2→L 信号控制,将 F_{i-1} 送到 L_i;直传即不移位,用 F→L 信号控制,将 F_i 送到 L_i。

注意:移位器与移位寄存器不同,它本身只有移位功能,没有寄存功能,所以移位之后的结果一定要保存到有关寄存器中。

4.3.2 带符号数的舍入操作

在算术右移时,由于受到硬件的限制,运算结果有可能需要舍去一定的尾数,这会造成一些误差。为了缩小误差,就要进行舍入处理。假定经过运算后的数共有 $p+q$ 位,现仅允许保留前 p 位。舍入方法有许多种,常见的舍入方法有:

① 恒舍(切断)。这是一种最容易实现的舍入方法,无论多余部分 q 位为何代码,一律舍去,保留部分的 p 位不作任何改变。

② 冯·诺依曼舍入法。这种舍入法又称为恒置 1 法,即不论多余部分 q 位为何代码,都把保留部分 p 位的最低位置 1。

③ 下舍上入法。下舍上入就是 0 舍 1 入,相当于十进制中的四舍五入。用将要舍去的 q 位的最高位作为判断标志,以决定保留部分是否加 1。如果该位为 0,则舍去整个 q 位(相当于恒舍);如果该位为 1,则在保留的 p 位的最低位上加 1。

④ 查表舍入法。查表舍入法又称 ROM 舍入法,因为它用 ROM 来存放舍入处理表,每

次经查表来读得相应的处理结果。查表法的原理框图如图 4-10 所示，图中的 ROM 容量为 256×7 位。通常，ROM 表的容量为2^K个单元，每个单元字长为 $K-1$ 位。舍入处理表的内容设置一般采用的方法是：当 K 位数据的高 $K-1$ 位为全 1 时，让那些单元按恒舍法填入 $K-1$ 位为全 1，其余单元都按下舍上入法来填其内容。例如，4 位数经 ROM 查表，舍入成 3 位结果，其 ROM 的地址和内容的对应关系如表 4-2 所示。

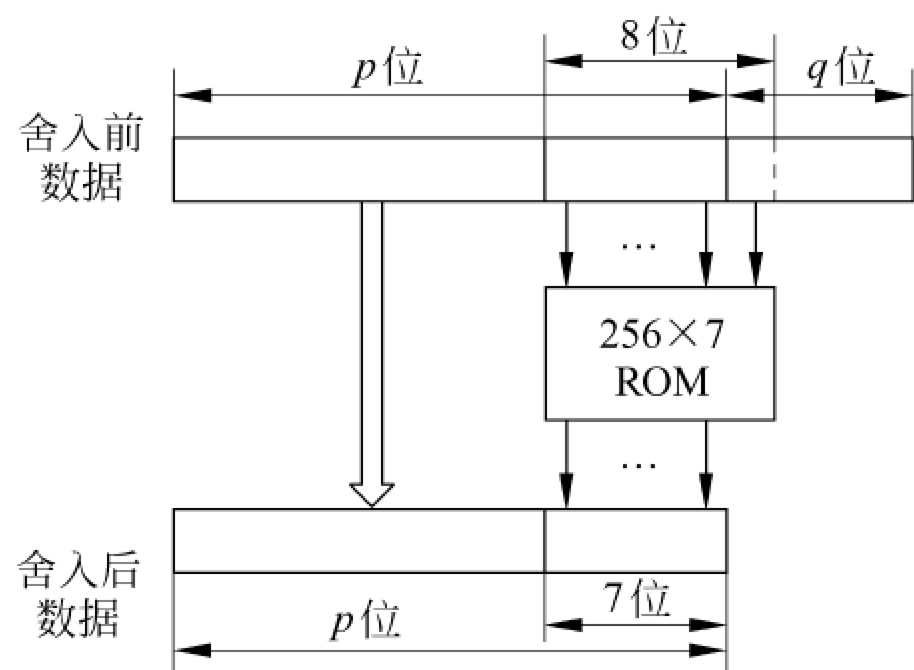


图 4-10　查表舍入法的原理

表 4-2　ROM 地址和内容的对应关系

地址	内容	地址	内容
0000	000	0100	010
0001	001	0101	011
0010	001	0110	011
0011	010	0111	100
1000	100	1100	110
1001	101	1101	111
1010	101	1110	111
1011	110	1111	111

4.4　定点乘法运算

在计算机中，乘法运算大多数由累加与移位来实现，也有些机器中具有由大规模集成电路制造的阵列乘法模块。

4.4.1　原码一位乘法

1. 原码一位乘法算法

原码一位乘法是从手算演变而来的，即用两个操作数的绝对值相乘，乘积的符号为两操作数符号的异或值（同号为正，异号为负），即：

$$\text{乘积}\quad P=|X|\times|Y|$$

$$\text{符号}\quad P_s=X_s\oplus Y_s$$

式中，P_s为乘积的符号，X_s和 Y_s为被乘数和乘数的符号。

例 4-7　设 $X=0.1101$，$Y=-0.1011$，列出手算乘法算式为：

```
          0.1 1 0 1     ---  被乘数
       ×  0.1 0 1 1     ---  乘数
       ----------------
            1 1 0 1     ---  部分积
          1 1 0 1       ---  部分积
        0 0 0 0         ---  部分积
    +  1 1 0 1          ---  部分积
    -------------------
     0.1 0 0 0 1 1 1 1  ---  乘积
```

因为　　　　$P_s = X_s \oplus Y_s = 0 \oplus 1 = 1$

所以　　　　$X \times Y = -0.10001111$

在手算乘法中,对应于每一位乘数求得一项部分积,然后将所有部分积一起相加求得最后乘积。然而,在计算机中实现原码乘法时,不能直接照搬上述方法。这是因为:

① 在加法器内很难实现多个数据同时相加。

② 加法器的位数一般与寄存器位数相同,而不是寄存器位数的两倍。

所以,在计算机中,通常把 n 位乘转化为 n 次"累加与移位"。每一次只求一位乘数所对应的新部分积,并与原部分积作一次累加;为了节省器件,用原部分积的右移来代替新部分积的左移。原码一位乘法的规则为:

① 参加运算的操作数取其绝对值。

② 令乘数的最低位为判断位,若为1,加被乘数;若为0,不加被乘数(加0)。

③ 累加后的部分积以及乘数右移一位。

④ 重复 n 次②和③。

⑤ 符号位单独处理,同号为正,异号为负。

通常,乘法运算需要3个寄存器。被乘数存放在B寄存器中;乘数存放在C寄存器中;A寄存器用来存放部分积与最后乘积的高位部分,它的初值为0。运算结束后寄存器C中不再保留乘数,改为存放乘积的低位部分。

例 4-8　已知 $X=0.1101$,$Y=-0.1011$,求 $X \times Y$。

$$|X| = 00.1101 \rightarrow B, \quad |Y| = .1011 \rightarrow C, \quad 0 \rightarrow A$$

	A	C	说明
	00.0000	1011	
$+\|X\|$	00.1101		$C_4=1$, $+\|X\|$
	00.1101		
→	00.0110	1 101	部分积右移一位
$+\|X\|$	00.1101		$C_4=1$, $+\|X\|$
	01.0011		
→	00.1001	11 10	部分积右移一位
+0	00.0000		$C_4=0$, +0
	00.1001		
→	00.0100	111 1	部分积右移一位
$+\|X\|$	00.1101		$C_4=1$, $+\|X\|$
	01.0001		
→	00.1000	1111	部分积右移一位

因为　　　　$P_s = X_s \oplus Y_s = 0 \oplus 1 = 1$

所以　　　　$X \times Y = -0.10001111$

原码一位乘法的流程图如图4-11所示,图中CR表示计数器,用来控制累加与移位的次数。

2. 原码一位乘法运算的实现

实现原码一位乘法运算器框图如图4-12所示。图中A、B是$n+2$位的寄存器,C是n位的寄存器,A寄存器和C寄存器是级联在一起的,它们都具有右移一位的功能,在右移控制信号的作用下,A寄存器最低一位的值将移入C寄存器的最高位。C寄存器的最低位的值作为字级与门的控制信号,以控制加被乘数还是不加被乘数(即加0)。C寄存器中的乘数在逐次右移过程中将逐步丢失,取而代之的是乘积的低位部分。原码一位乘法运算器电路中除去3个寄存器外,还需要一个$n+2$位的加法器、一个计数器、$n+2$个与门(控制是否加被乘数)和一个异或门(处理符号位)。

图4-11　原码一位乘法流程图

图4-12　原码一位乘法运算器

4.4.2　补码一位乘法

虽然原码乘法比补码乘法容易实现,但因为补码加减法简单,在以加减运算为主的通用机中操作数都用补码表示,所以这类计算机在做乘法时常使用补码乘法。

1. 校正法

补码乘法不能简单地套用原码乘法的算法,这是因为补码的符号位是参加运算的。所谓校正法,是将$[X]_补$和$[Y]_补$按原码规则运算,所得结果根据情况再加以校正,从而得到正确的$[X\times Y]_补$。

① 当乘数$Y>0$时,不管被乘数X的正负都直接按原码乘法运算,只是移位时按补码规则进行。

② 当乘数$Y<0$时,可以先把$[Y]_补$的符号位丢掉不管,仍按原码乘法运算,最后再加上$[-X]_补$进行校正。

将上述两种情况综合起来,就得到了补码乘法的统一表达式:

$$[X\times Y]_{补}=[X]_{补}\times(0.Y_1Y_2\cdots Y_n)+[-X]_{补}\times Y_s$$

2. 比较法——Booth 乘法

校正法在乘数为负数的情况下,需要进行校正,控制起来要复杂一些,希望有一个对于正数和负数都一致的算法,这就是比较法。比较法是英国的 Booth 夫妇提出来的,因此又称为 Booth 法。

假设被乘数$[X]_{补}=X_s.X_1X_2\cdots X_n$,乘数$[Y]_{补}=Y_s.Y_1Y_2\cdots Y_n$。

根据校正法的统一表达式,有:

$$\begin{aligned}[X\times Y]_{补}&=[X]_{补}\times(0.Y_1Y_2\cdots Y_n)+[-X]_{补}\times Y_s\\&=[X]_{补}\times(Y_12^{-1}+Y_22^{-2}+\cdots+Y_n2^{-n})+[-X]_{补}\times Y_s\\&=[X]_{补}\times\{-Y_s+(Y_1-Y_12^{-1})+(Y_22^{-1}-Y_22^{-2})+\cdots\\&\quad+(Y_n2^{-(n-1)}-Y_n2^{-n})+0\}\\&=[X]_{补}\times\{(Y_1-Y_s)+(Y_2-Y_1)2^{-1}+\cdots+(0-Y_n)2^{-n}\}\\&=[X]_{补}\times\{(Y_1-Y_s)+(Y_2-Y_1)2^{-1}+\cdots+(Y_{n+1}-Y_n)2^{-n}\}\end{aligned}$$

式中,Y_s代表符号位,Y_{n+1}是附加位,它的初值为0,增加附加位不会影响运算结果。根据上式可写出递推公式:

$$\begin{aligned}&[Z_0]_{补}=0\\&[Z_1]_{补}=2^{-1}\{[Z_0]_{补}+(Y_{n+1}-Y_n)[X]_{补}\}\\&[Z_2]_{补}=2^{-1}\{[Z_1]_{补}+(Y_n-Y_{n-1})[X]_{补}\}\\&\qquad\vdots\\&[Z_n]_{补}=2^{-1}\{[Z_{n-1}]_{补}+(Y_2-Y_1)[X]_{补}\}\end{aligned}$$

所以

$$[X\times Y]_{补}=[Z_n]_{补}+(Y_1-Y_s)[X]_{补}$$

式中,$[Z_0]_{补}$为初始部分积,$[Z_1]_{补}\sim[Z_n]_{补}$依次为各次求得的累加并右移之后的部分积。

由上式可以发现,每次运算取决于乘数相邻两位Y_i、Y_{i+1}的值,把它们称为乘法的判断位。这种运算是根据乘数相邻两位的比较结果$(Y_{i+1}-Y_i)$来确定运算操作,因此称为比较法。

Booth 乘法规则如下:

① 参加运算的数用补码表示。

② 符号位参加运算。

③ 乘数最低位后面增加一位附加位Y_{n+1},其初值为0。

④ 由于每求一次部分积要右移一位,所以乘数的最低两位Y_n、Y_{n+1}的值决定了每次应执行的操作,如表4-3所示。

表 4-3 Booth 乘法运算操作

判断位 $Y_n\ Y_{n+1}$	操 作	判断位 $Y_n\ Y_{n+1}$	操 作
0 0	原部分积+0,右移一位	1 0	原部分积+$[-X]_{补}$,右移一位
0 1	原部分积+$[X]_{补}$,右移一位	1 1	原部分积+0,右移一位

⑤ 移位按补码右移规则进行。

⑥ 共需做 $n+1$ 次累加，n 次移位，第 $n+1$ 次不移位。

注意：由于符号位要参加运算，部分积累加时最高有效位产生的进位可能会侵占符号位，故被乘数和部分积应取双符号位，而乘数只需要一位符号位。运算时仍需要有 3 个寄存器，各自的作用与原码时相同，只不过存放的内容均为补码而已。

例 4-9 已知 $X=-0.1101$，$Y=0.1011$，求 $X\times Y$。

$[X]_{补}=11.0011\rightarrow \mathrm{B}$， $[Y]_{补}=0.1011\rightarrow \mathrm{C}$， $0\rightarrow \mathrm{A}$

$[-X]_{补}=00.1101$

	A	C 附加位↓	说明
	00.0000	0.101**10**	
+[−X]补	00.1101		$C_4C_5=10$，+[−X]补
	00.1101		
→	00.0110	1|010**11**	部分积右移一位
+0	00.0000		$C_4C_5=11$，+0
	00.0110		
→	00.0011	01|01**01**	部分积右移一位
+[X]补	11.0011		$C_4C_5=01$，+[X]补
	11.0110		
→	11.1011	001|0**10**	部分积右移一位
+[−X]补	00.1101		$C_4C_5=10$，+[−X]补
	00.1000		
→	00.0100	0001|**01**	部分积右移一位
+[X]补	11.0011		$C_4C_5=01$，+[X]补
	11.0111		

所以 $[X\times Y]_{补}=1.01110001$

$X\times Y=-0.10001111$

Booth 乘法的流程如图 4-13 所示。

3. Booth 乘法运算的实现

实现 Booth 乘法的运算器框图如图 4-14 所示。各器件的作用与原码一位乘法相同，A 和 B寄存器长 $n+2$ 位，C 寄存器也有 $n+2$ 位，还需一个 $n+2$ 位的加法器、$n+2$ 个与或门和一个计数器。由 C 寄存器的最低两位 C_nC_{n+1} 来控制是加/减被乘数还是加 0，当 $C_nC_{n+1}=01$ 时，加被乘数，即加 B 寄存器的内容；$C_nC_{n+1}=10$ 时，减被乘数，即加上 B 寄存器中内容的反，并在加法器的最低位加 1；$C_nC_{n+1}=00$ 或 11 时，不加也不减（加 0）。由于符号位参与运算，所以不需要专门处理符号位的异或门。

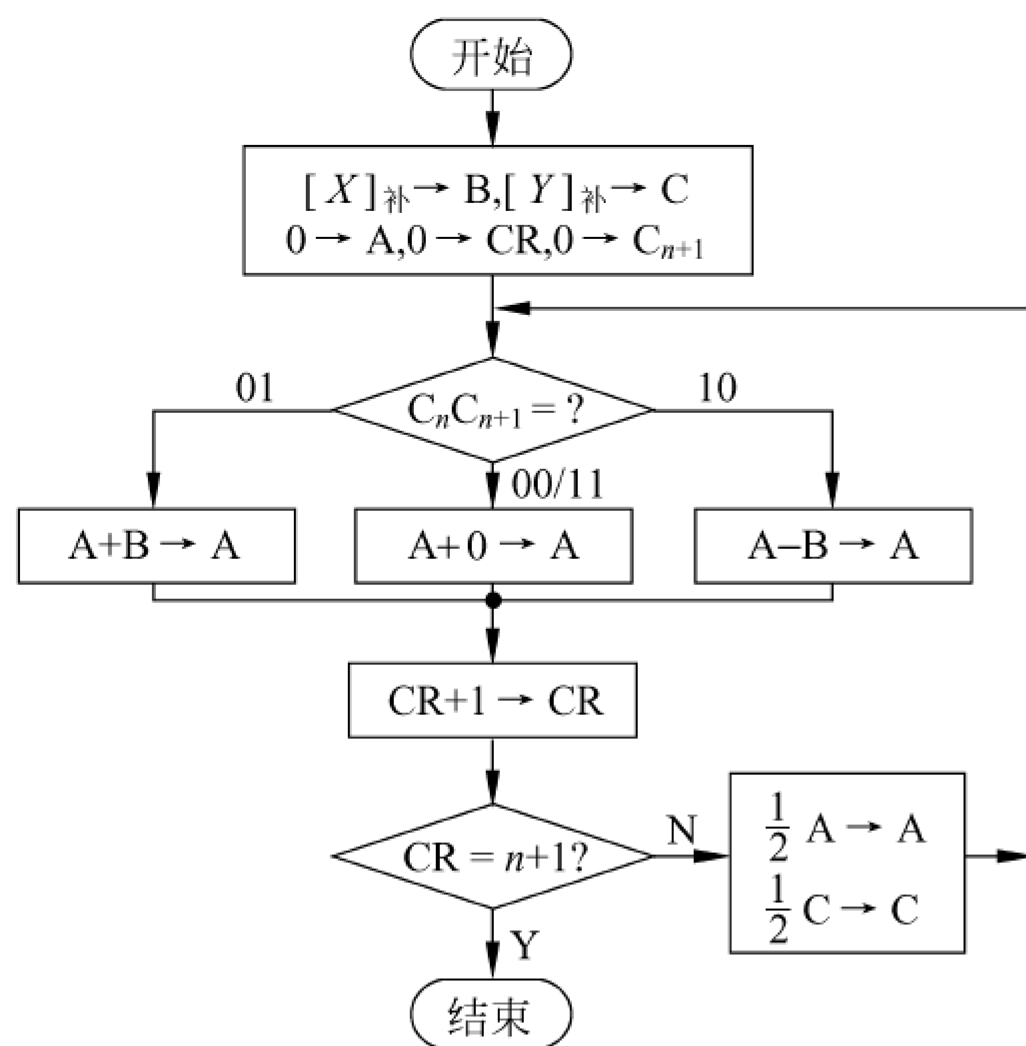


图 4-13　Booth 乘法流程图

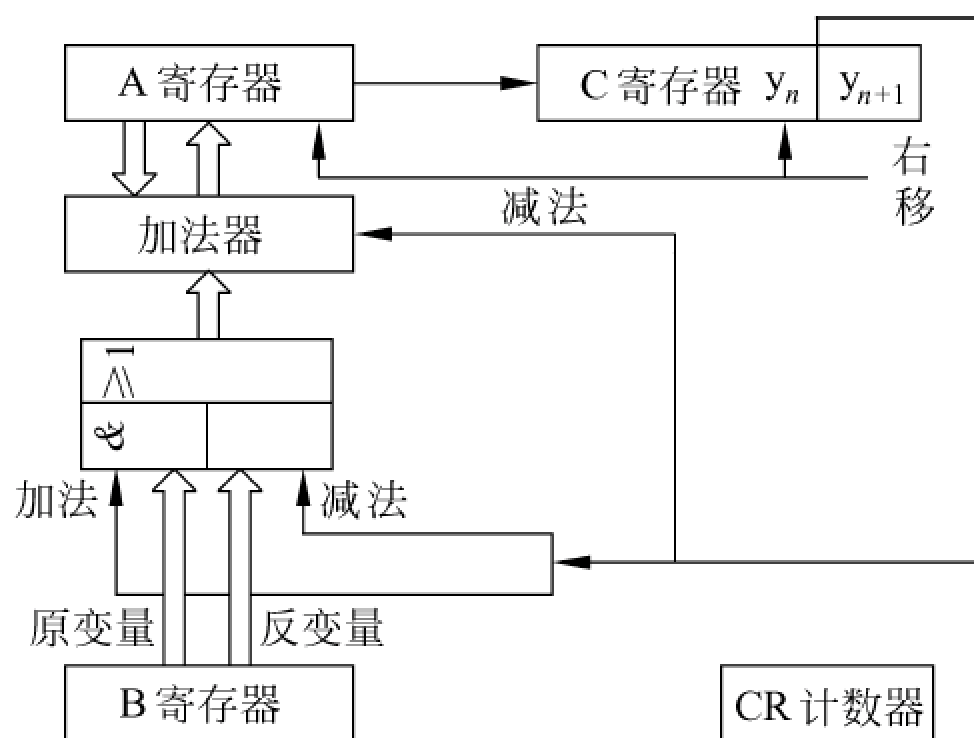


图 4-14　Booth 乘法运算器

4.4.3　补码两位乘法

为了提高乘法的执行速度，可以选用两位乘法的方案。所谓两位乘法，就是每次处理乘数中的两位，从而使乘法的速度提高了一倍。为了简单起见，这里只介绍补码两位乘法。

可以根据前面介绍的 Booth 乘法方便地推导出补码两位乘法，即把补码两位乘理解为将 Booth 乘法的两次合并为一次来做。

假定上次乘法的部分积表示为$[Z']_补$，本次的部分积表示为$[Z'']_补$，则有：

$$[Z']_补 = 2^{-1}\{[Z]_补 + (Y_{i+1} - Y_i)\,[X]_补\}$$

$$\begin{aligned}[Z'']_补 &= 2^{-1}\{[Z']_补 + (Y_i - Y_{i-1})\,[X]_补\}\\ &= 2^{-1}\{2^{-1}\{[Z]_补 + (Y_{i+1} - Y_i)\,[X]_补\} + (Y_i - Y_{i-1})\,[X]_补\}\\ &= 2^{-2}\{[Z]_补 + (Y_{i+1} - Y_i)\,[X]_补 + 2(Y_i - Y_{i-1})[X]_补\}\\ &= 2^{-2}\{[Z]_补 + (Y_{i+1} + Y_i - 2Y_{i-1})\,[X]_补\}\end{aligned}$$

由上式可见，补码两位乘法可以通过 $Y_{i-1}\ Y_i\ Y_{i+1}$ 3 位作为判断位来确定运算操作。补码两位乘法规则中除④和⑥外，其余都与 Booth 乘法规则相同。

补码两位乘法根据乘数的最低 3 位 $Y_{n-1}Y_n\ Y_{n+1}$ 的值（做 $Y_{n+1}+Y_n-2Y_{n-1}$）决定每次应执行的操作，如表 4-4 所示。

表 4-4　补码两位乘法操作

判断位 $Y_{n-1}Y_n\ Y_{n+1}$	操　　作
0　0　0	原部分积+0，右移两位
0　0　1	原部分积+$[X]_补$，右移两位
0　1　0	原部分积+$[X]_补$，右移两位
0　1　1	原部分积+$2[X]_补$，右移两位
1　0　0	原部分积+$2[-X]_补$，右移两位
1　0　1	原部分积+$[-X]_补$，右移两位
1　1　0	原部分积+$[-X]_补$，右移两位
1　1　1	原部分积+0，右移两位

被乘数和部分积取 3 个符号位，当乘数的数值位 n 为偶数时，乘数取 2 个符号位，共需作 $\frac{n}{2}+1$ 次累加，$\frac{n}{2}$ 次移位（最后一次不移位）；当 n 为奇数时，乘数只需 1 个符号位，共需 $\frac{n+1}{2}$ 次累加和移位，但最后一次仅移一位。

例 4-10　已知 $X=0.0110011$，$Y=-0.0110010$，求 $X\times Y$。

$[X]_补=000.0110011\to B$，$[Y]_补=1.1001110\to C$，$0\to A$。

$2[X]_补=000.1100110$，$[-X]_补=111.1001101$，$2[-X]_补=111.0011010$。

	A	C　附加位	说明
	000.0000000	1.10011**100**	
$+2[-X]_补$	111.0011010		$C_6C_7C_8=100$，$+2[-X]_补$
	111.0011010		
2→	111.1100110	10|1100**111**	部分积右移两位
+0	000.0000000		$C_6C_7C_8=111$，+0
	111.1100110		
2→	111.1111001	1010|11**001**	部分积右移两位
$+[X]_补$	000.0110011		$C_6C_7C_8=001$，$+[X]_补$
	000.0101100		
2→	000.0001011	001010|**110**	部分积右移两位
$+[-X]_补$	111.1001101		$C_6C_7C_8=110$，$+[-X]_补$
	111.1011000		
1→	111.1101100	0001010|**11**	最后一次右移一位

所以 $[X\times Y]_{补}=1.11011000001010$

$X\times Y=-0.00100111110110$

至此,介绍了原码、补码一位乘法和补码两位乘法,对于初学者来说,往往会在运算次数、符号位取多少位、符号位是否参加运算等问题上出错,为了帮助记忆,特将这3种乘法运算的上述问题统一列于表4-5中。

表4-5 乘法运算总结

<table>
<tr><th rowspan="2">乘法类型</th><th colspan="3">符号位</th><th rowspan="2">累加次数</th><th colspan="3">移位</th></tr>
<tr><th>参与运算</th><th>部分积</th><th>乘数</th><th>方向</th><th>次数</th><th>每次位数</th></tr>
<tr><td>原码一位乘法</td><td>否</td><td>2</td><td>0</td><td>n</td><td>右</td><td>n</td><td>1</td></tr>
<tr><td>补码一位乘法</td><td>是</td><td>2</td><td>1</td><td>$n+1$</td><td>右</td><td>n</td><td>1</td></tr>
<tr><td rowspan="2">补码两位乘法</td><td rowspan="2">是</td><td rowspan="2">3</td><td>2(n为偶数)</td><td>$\frac{n}{2}+1$</td><td>右</td><td>$\frac{n}{2}$</td><td>2</td></tr>
<tr><td>1(n为奇数)</td><td>$\frac{n+1}{2}$</td><td>右</td><td>$\frac{n+1}{2}$</td><td>2(最后一次移一位)</td></tr>
</table>

注:n为乘数的数值部分的位数。

4.4.4 阵列乘法器

为了进一步提高乘法运算的速度,可采用高速乘法模块组成的阵列乘法器。设有两个无符号的二进制整数:

$$A=\sum_{i=0}^{m-1}a_i\times 2^i,\quad B=\sum_{j=0}^{n-1}b_j\times 2^j$$

所以

$$P=A\times B=\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(a_i\times b_j)\times 2^{(i+j)}=\sum_{k=0}^{m+n-1}P_k\times 2^k$$

例4-11 在上面公式中当$m=n=5$时,则有:

$$\begin{array}{ccccccccccc}
 & & & & & a_4 & a_3 & a_2 & a_1 & a_0 & =A \\
 & & & & \times & b_4 & b_3 & b_2 & b_1 & b_0 & =B \\
\hline
 & & & & & a_4b_0 & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 & \\
 & & & & a_4b_1 & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 & & \\
 & & & a_4b_2 & a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 & & & \\
 & & a_4b_3 & a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 & & & & \\
+ & a_4b_4 & a_3b_4 & a_2b_4 & a_1b_4 & a_0b_4 & & & & & \\
\hline
P_9 & P_8 & P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0 & =P
\end{array}$$

图4-15所示为5×5位绝对值相乘的阵列乘法器原理图。其中,FA表示全加器,虚线框中是具有并行进位链的并行加法器。这种结构可同时得到各项部分积,并且一次将其相加就能得到乘积,因此运算速度很快。

若采用补码相乘时,可在上述乘法阵列外再使用3个求补器,其中两个算前求补器将两个操作数在相乘之前先变成正整数,而一个算后求补器是当结果为负(即两个操作数的符号不一致)时把运算结果变换成补码。

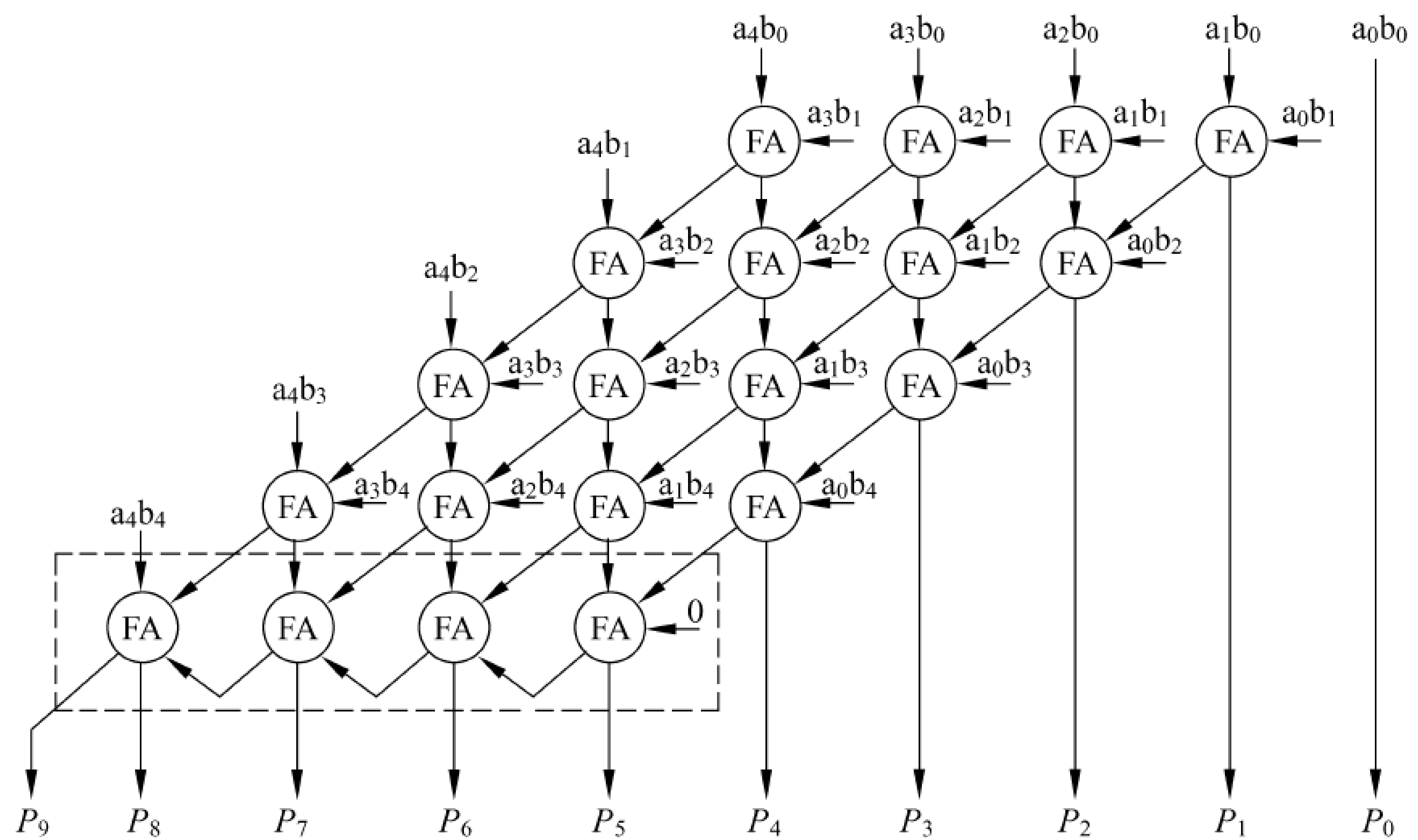


图 4-15　5×5 位绝对值相乘的阵列乘法器原理

4.5　定点除法运算

除法是乘法的逆运算，与乘法运算的处理思想相似，可以将 n 位除转化成若干次“减法-移位”，也有些计算机具有由大规模集成电路制造的阵列除法模块。

4.5.1　原码除法运算

1. 原码比较法和恢复余数法

(1) 比较法

先看手工除法的计算过程。

假设 $X=0.1011$，$Y=0.1101$，则有：

```
                0.1 1 0 1      --- 商
0.1 1 0 1 / 0.1 0 1 1 0        --- 被除数
            0.0 1 1 0 1
            ---------------
            0.0 1 0 0 1 0      --- 部分余数
            0.0 0 1 1 0 1
            ---------------
            0.0 0 0 1 0 1 0 0  --- 部分余数
            0.0 0 0 0 1 1 0 1
            ---------------
            0.0 0 0 0 0 1 1 1  --- 余数
```

因为　$$Q_s=X_s\oplus Y_s=0\oplus 0=0$$

所以　$$X\div Y=0.1101+\frac{0.0111\times 2^{-4}}{0.1101}$$

手工计算的规则是：首先判断被除数(或部分余数)和除数的大小，若除数小于或等于被除数(或部分余数)的最高几位，就将该位商上"1"，并从被除数(或部分余数)中减去除数，得到新的余数；若除数大于被除数(或部分余数)，就将该位商上"0"，被除数(或部分余数)不变。然后，将被除数的下一位挪下来(若存在)或在部分余数的最低位补"0"，再与除数进行比较，直至除尽或得到的商的位数满足要求为止。

比较法类似于手工运算，只是为了便于机器操作，将除数右移改为部分余数左移，每一位的上商直接写到寄存器的最低位。设A寄存器中存放被除数(或部分余数)，B寄存器中存放除数，C寄存器用来存放商Q，若$A \geqslant B$，则上商"1"，并减除数；若$A < B$，则上商"0"。比较过程的流程如图4-16(a)所示。比较法需要设置比较线路，从而增加了硬件的代价。

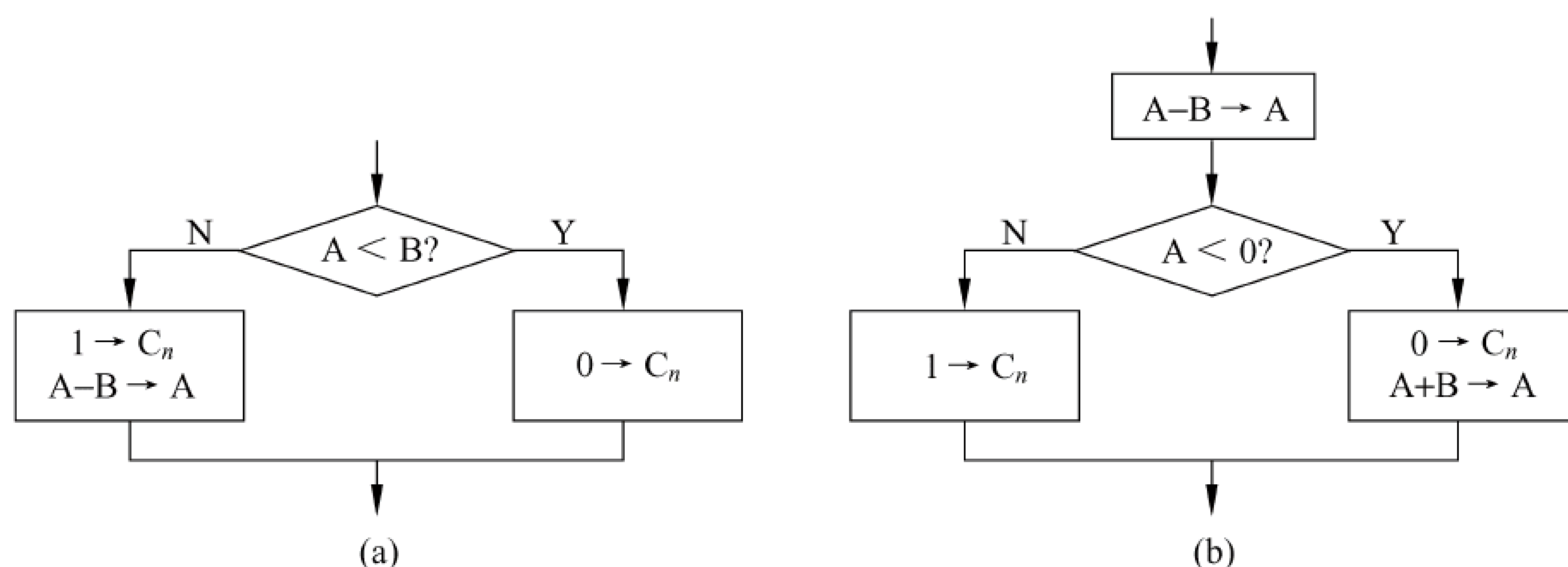


图4-16　比较和恢复余数过程流程图

(2) 恢复余数法

恢复余数法是直接做减法试探方法，不管被除数(或部分余数)减除数是否够减，都一律先做减法。若部分余数为正，表示够减，则该位商上"1"；若部分余数为负，表示不够减，则该位商上"0"，并要恢复余数。恢复余数过程的流程如图4-16(b)所示。

由于部分余数的正、负是根据不同的操作数组合随机出现的，恢复除数法会使得除法运算的实际操作次数不固定，从而导致控制电路比较复杂。而且在恢复余数时，要多做一次加法，从而降低了除法的执行速度。因此，原码恢复余数法在计算机中一般很少采用。

2. 原码不恢复余数法(原码加减交替法)

原码不恢复余数法是对恢复余数法的一种改进，它减少了浪费的加法时间，且运算的次数固定，故被广泛采用。

在恢复余数法中，若第$i-1$次求商的部分余数为r_{i-1}，则第i次求商操作为$r_i = 2r_{i-1} - Y$。

若够减，部分余数$r_i = 2r_{i-1} - Y > 0$，商1。

若不够减，部分余数$r_i = 2r_{i-1} - Y < 0$，商0，恢复余数后，$r_i' = r_i + Y = 2r_{i-1}$，然后再左移一位，进行第$i+1$次操作：

$$r_{i+1} = 2r_i' - Y = 2(r_i + Y) - Y = 2r_i + 2Y - Y = 2r_i + Y$$

上式表明，当出现不够减的情况下并不需要恢复余数，可以直接做下一次操作，但操作是$2r_i + Y$，其结果与恢复余数后左移一位再减Y是等效的。因此，原码不恢复余数除法的

规则可由下面的通式表示：

$$r_{i+1}=2r_i+(1-2Q_i)\times Y$$

式中 Q_i 为第 i 次所得的商，若部分余数为正，则 $Q_i=1$，部分余数左移一位，下一次继续减除数；若部分余数为负，则 $Q_i=0$，部分余数左移一位，下一次加除数。由于加减运算交替地进行，故称为原码加减交替法。

除法运算需要 3 个寄存器。A 寄存器和 B 寄存器分别用来存放被除数和除数，C 寄存器用来存放商，它的初值为 0。运算过程中 A 寄存器的内容为部分余数，它将不断地变化，最后剩下的是扩大了若干倍的余数，只有将它乘上 2^{-n} 才是真正的余数。

例 4-12 已知 $X=-0.10101$，$Y=0.11110$，求 $X\div Y$。

$$\lvert X\rvert=00.10101\rightarrow A,\quad \lvert Y\rvert=00.11110\rightarrow B,\quad 0\rightarrow C$$

$$[\lvert Y\rvert]_{变补}=11.00010$$

	A	C	说明
	00.10101	0.00000	
$+[\lvert Y\rvert]_{变补}$	11.00010		$-\lvert Y\rvert$
	11.10111	0.0000**<u>0</u>**	部分余数为负，商 0
←	11.01110		左移一位
$+\lvert Y\rvert$	00.11110		$+\lvert Y\rvert$
	00.01100	0.000**0<u>1</u>**	部分余数为正，商 1
←	00.11000		左移一位
$+[\lvert Y\rvert]_{变补}$	11.00010		$-\lvert Y\rvert$
	11.11010	0.00**01<u>0</u>**	部分余数为负，商 0
←	11.10100		左移一位
$+\lvert Y\rvert$	00.11110		$+\lvert Y\rvert$
	00.10010	0.0**010<u>1</u>**	部分余数为正，商 1
←	01.00100		左移一位
$+[\lvert Y\rvert]_{变补}$	11.00010		$-\lvert Y\rvert$
	00.00110	0.**0101<u>1</u>**	部分余数为正，商 1
←	00.01100		左移一位
$+[\lvert Y\rvert]_{变补}$	11.00010		$-\lvert Y\rvert$
	11.01110	**0.10110**	部分余数为负，商 0
$+\lvert Y\rvert$	00.11110		最后一次恢复余数，$+\lvert Y\rvert$
	00.01100		

原码除法和原码乘法一样，符号位是单独处理的。所以，

$$Q_s=X_s\oplus Y_s=1\oplus 0=1$$

$$\frac{X}{Y}=-\left(0.10110+\frac{0.01100\times 2^{-5}}{0.11110}\right)$$

原码加减交替除法运算的算法流程图如图 4-17 所示。

注意：在定点小数除法运算时，为了防止溢出，要求被除数的绝对值小于除数的绝对值，$\lvert X\rvert<\lvert Y\rvert$（$\lvert X\rvert=\lvert Y\rvert$ 除外），且除数不能为 0。因此，第一次减除数肯定是不够减的，如果采用先移位后减除数的方法，得到的结果也是相同的。另外，在原码加减交替法中，当最终余数为负数时，必须恢复一次余数，使之变为正余数，此时则不需要再左移了。

3. 原码加减交替除法的实现

实现原码加减交替法的运算器框图如图4-18所示,A、B寄存器长$n+2$位,C寄存器长$n+1$位,还需一个$n+2$位的加法器、$n+2$个与或门、一个计数器和一个异或门。其中,A寄存器和C寄存器是级联在一起的,它们都具有左移一位的功能,在左移控制信号的作用下,C寄存器最高位的值将移入A寄存器的最低位。A寄存器中的初值是被除数,但在运算过程中将变为部分余数。C寄存器的最低位用来保存每次运算得到的商值,此商值同时也作为下一次操作是做加法还是做减法的控制信号。

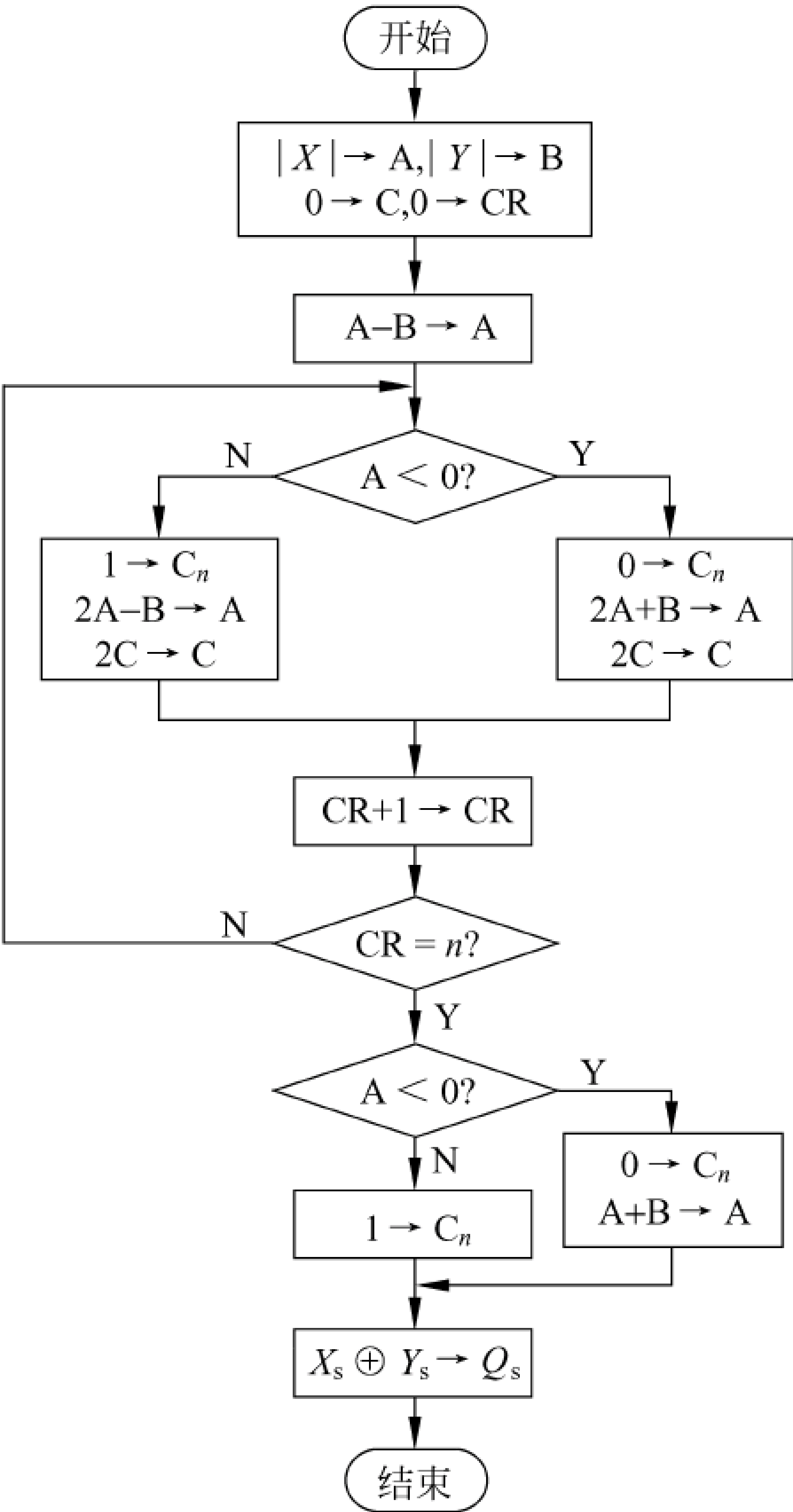


图4-17　原码加减交替除法流程图

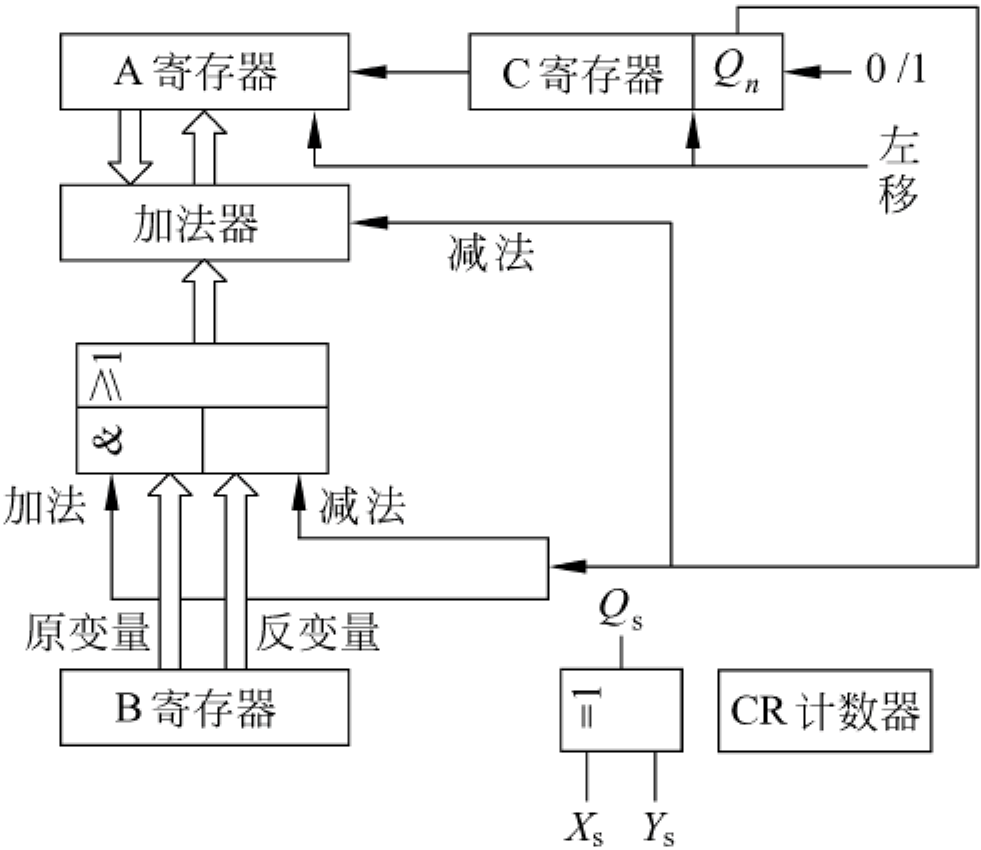


图4-18　原码加减交替法运算器框图

4.5.2　补码除法运算

被除数和除数都用补码表示,符号位参加运算,商和余数也用补码表示,运算时应考虑以下问题。

1. 够减的判断

参加运算的两个数符号任意,当被除数(或部分余数)的绝对值大于或等于除数的绝对值时称为够减,反之称为不够减。为了判断是否够减,当两数同号时,实际应做减法;两数异

号时,实际应做加法。

判断的方法和结果如下:当被除数(或部分余数)与除数同号时,如果得到的新部分余数与除数同号则表示够减,否则为不够减;当被除数(或部分余数)与除数异号时,如果得到的新部分余数与除数异号则表示够减,否则为不够减。

2. 上商规则

补码除法运算的商也是用补码表示的,上商的规则是:如果$[X]_补$和$[Y]_补$同号,则商为正数,够减时上商“1”,不够减时上商“0”;如果$[X]_补$和$[Y]_补$异号,则商为负数,够减时上商“0”,不够减时上商“1”。

将上商规则与够减的判断结合起来,可得到商的确定方法,如表4-6所示。

表4-6 商的确定方法

$[X]_补$与$[Y]_补$	商	$[r_i]_补$与$[Y]_补$	上商
同号	正	同号,表示够减	1
		异号,表示不够减	0
异号	负	异号,表示够减	0
		同号,表示不够减	1

从表4-6中可看出,补码的上商规则可归结为:若部分余数$[r_i]_补$和除数$[Y]_补$同号,则商上“1”;反之则商上“0”。

3. 商符的确定

商符是在求商的过程中自动形成的,按补码上商规则,第一次得出的商就是实际应得的商符。为了防止溢出,必须有$|X|<|Y|$,所以第一次肯定不够减。当被除数与除数同号时,部分余数与除数必然异号,商上“0”,恰好与商符一致;当被除数与除数异号,部分余数与除数必然同号,商上“1”,也恰好就是商的符号。

4. 求新部分余数

求新部分余数$[r_{i+1}]_补$的通式如下:

$$[r_{i+1}]_补=2[r_i]_补+(1-2Q_i)\times[Y]_补$$

式中,Q_i表示第i步的商。若商上“1”,则下一步操作为部分余数左移一位,减去除数;若商上“0”,则下一步操作为部分余数左移一位,加上除数。

补码加减交替法规则概括列于表4-7中。

表4-7 补码加减交替法规则

$[X]_补$与$[Y]_补$	第一次操作	$[r_i]_补$与$[Y]_补$	上商	下一次操作
同号	$[X]_补-[Y]_补$	① 同号(够减)	1	$[r_{i+1}]_补=2[r_i]_补-[Y]_补$
		② 异号(不够减)	0	$[r_{i+1}]_补=2[r_i]_补+[Y]_补$
异号	$[X]_补+[Y]_补$	① 同号(不够减)	1	$[r_{i+1}]_补=2[r_i]_补-[Y]_补$
		② 异号(够减)	0	$[r_{i+1}]_补=2[r_i]_补+[Y]_补$

5. 末位恒置1

假设商的数值位为 n 位，运算次数为 $n+1$ 次，商的最末一位恒置为“1”，运算的最大误差为 2^{-n}。此法操作简单，易于实现，在对商的精度没有特殊要求的情况下是一种简单实用的方法。

例 4-13 已知 $X=0.1000$，$Y=-0.1010$，求 $X \div Y$。

$[X]_补=00.1000 \rightarrow A$，$[Y]_补=11.0110 \rightarrow B$，$0 \rightarrow C$

$[-Y]_补=00.1010$

	A	C	说明
	00.1000	0.0000	
$+[Y]_补$	11.0110		$[X]_补$、$[Y]_补$异号，$+[Y]_补$
	11.1110	0.000**1**	$[r_i]_补$、$[Y]_补$同号，商1
←	11.1100		左移一位
$+[-Y]_补$	00.1010		$+[-Y]_补$
	00.0110	0.00**10**	$[r_i]_补$、$[Y]_补$异号，商0
←	00.1100		左移一位
$+[Y]_补$	11.0110		$+[Y]_补$
	00.0010	0.0**100**	$[r_i]_补$、$[Y]_补$异号，商0
←	00.0100		左移一位
$+[Y]_补$	11.0110		$+[Y]_补$
	11.1010	0.**1001**	$[r_i]_补$、$[Y]_补$同号，商1
←	11.0100		左移一位
$+[-Y]_补$	00.1010		$+[-Y]_补$
	11.1110	**1.0011**	末位恒置1

所以

$$\left[\frac{X}{Y}\right]_补=1.0011+\frac{1.1110\times 2^{-4}}{1.0110}$$

$$\frac{X}{Y}=-0.1101+\frac{-0.0010\times 2^{-4}}{-0.1010}=-0.1101+\frac{0.0010\times 2^{-4}}{0.1010}$$

补码加减交替除法的算法流程图如图 4-19 所示。

实现补码加减交替法的运算器框图与图 4-18 基本相似，只是加减和上商的条件不同，不需要异或门来处理符号位而已。

注意：无论在原码加减交替法还是补码加减交替法的左移过程中，都可能出现左移后双符号位不一致的情况，这是没有关系的，不会影响最后的运算结果，因为此时真符(最左边的一位符号位)并没有发生变化。

至此，介绍了原码、补码一位除法，为了帮助记忆，特将常用的原码、补码加减交替法的运算次数、符号位等问题统一列于表 4-8 中。

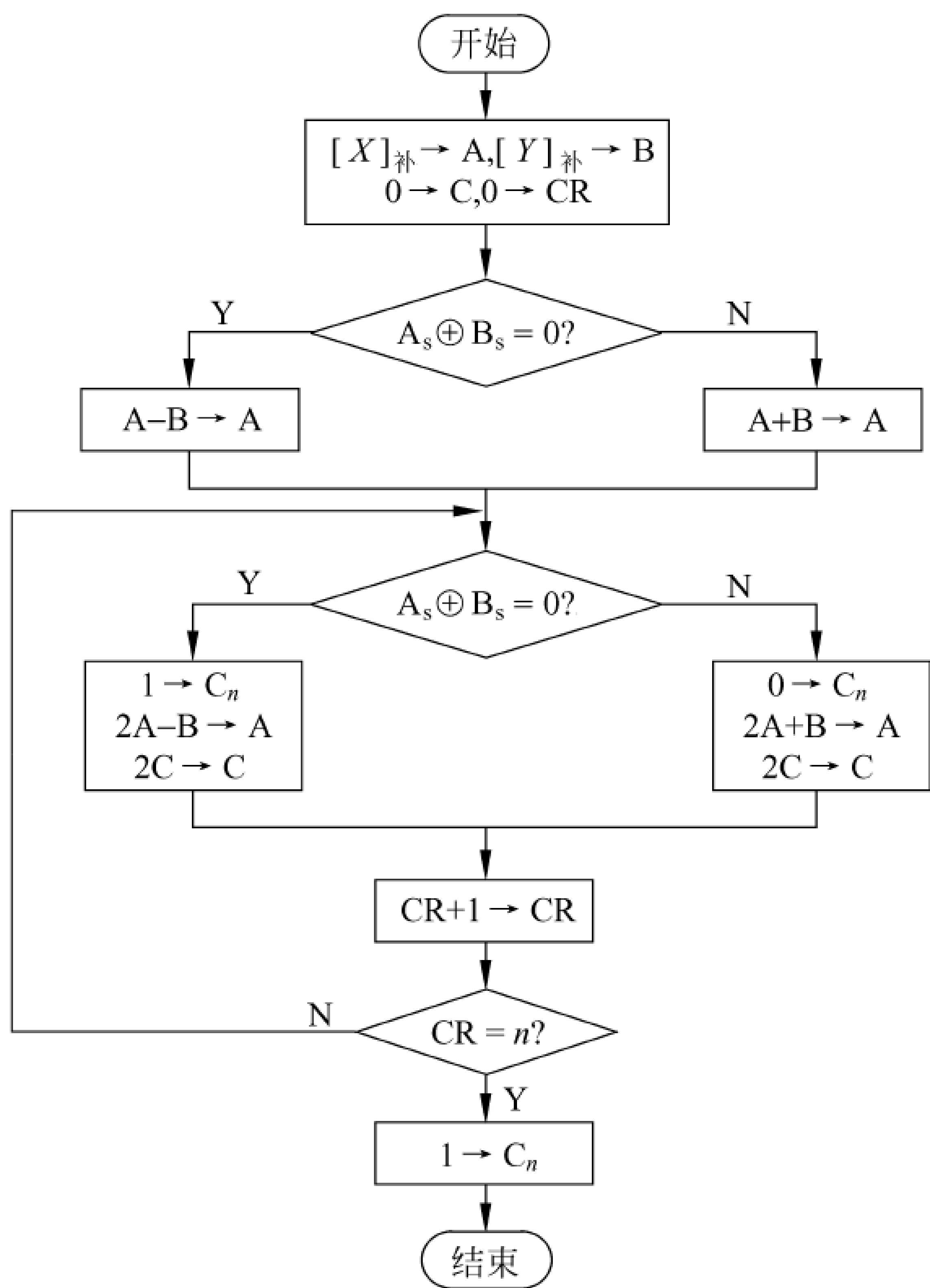


图 4-19　补码加减交替除法的算法流程图

表 4-8　除法运算总结

除法类型	符号位参与运算	加减次数	移位		备注
			方向	次数	
原码加减交替法	否	$n+1$ 或 $n+2$	左	n	若最终余数为负,需恢复余数
补码加减交替法	是	$n+1$	左	n	末位恒置 1

注：n 为除数的数值部分的位数。

4.5.3　阵列除法器

和阵列乘法器相似,阵列除法器也是一种并行运算部件,能够实现高速的除法运算。图 4-20 是一个实现加减交替除法的阵列除法器。设：被除数 $X=0.X_1X_2X_3X_4X_5X_6$,除数 $Y=0.Y_1Y_2Y_3$,商 $Q=0.Q_1Q_2Q_3$,余数 $r=0.00r_3r_4r_5r_6$。图中的每一个方框为一个可控加法和减法(CAS)单元,当其输入控制端等于 0 时,CAS 做加法运算;当输入控制端等于 1 时,CAS 做减法运算。

在除法阵列中,每一行所执行的操作究竟是加法还是减法,取决于前一行输出的符号与被除数的符号是否一致。当出现不够减时,部分余数相对于被除数来说要改变符号,这时应该产生商“0”。除数首先沿对角线右移,然后加到下一行的部分余数上。当部分余数不改变

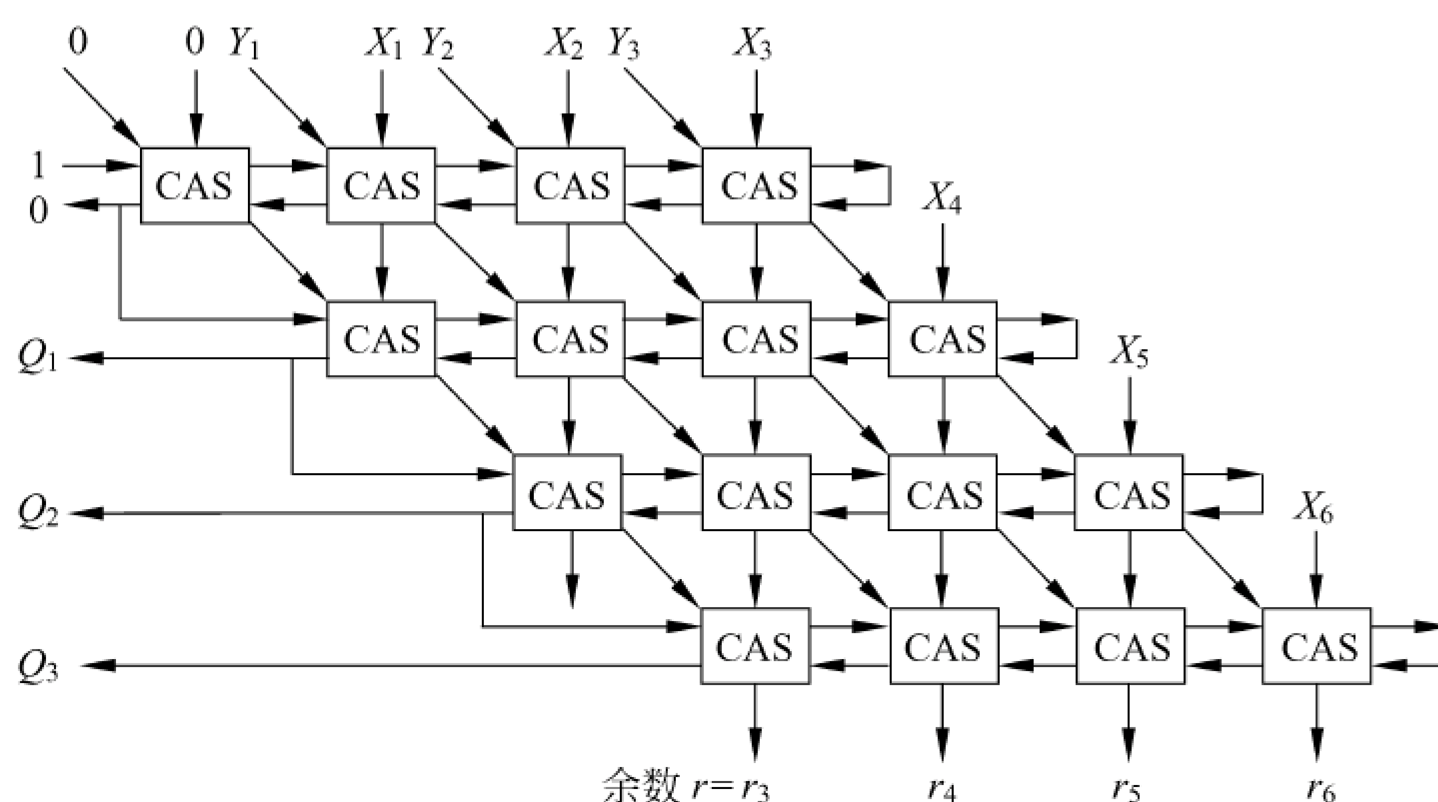


图 4-20 阵列除法器原理

它的符号时,即产生商“1”,下一行操作应该是减法。

4.6 规格化浮点运算

第 2 章中已经讨论了浮点数的表示方法,这里将进一步讨论规格化浮点数的四则运算问题,其中尾数的基数 $r=2$。

4.6.1 浮点加减运算

设两个非 0 的规格化浮点数分别为:

$$A=M_A\times 2^{E_A}$$

$$B=M_B\times 2^{E_B}$$

规格化浮点数 A、B 加减运算通式为:

$$A\pm B=(M_A,E_A)\pm(M_B,E_B)=\begin{cases}(M_A\pm M_B\times 2^{-(E_A-E_B)},E_A) & E_A>E_B\\(M_A\times 2^{-(E_B-E_A)}\pm M_B,E_B) & E_A<E_B\end{cases}$$

式中,$2^{-(E_A-E_B)}$ 和 $2^{-(E_B-E_A)}$ 称为移位因子。

1. 浮点数加减运算步骤

执行浮点数的加减运算,需要经过对阶、尾数加/减、尾数结果规格化、舍入、溢出判断等步骤。

(1) 对阶

两个浮点数相加或相减,首先要把小数点的位置对齐,而浮点数的小数点的实际位置取决于阶码的大小,因此,对齐两数的小数点就是使两数的阶码相等,这个过程称为对阶。

要对阶,首先应求出两数阶码 E_A 和 E_B 之差,即:

$$\Delta E=E_A-E_B$$

若 $\Delta E=0$,则表示两数阶码相等,即 $E_A=E_B$;若 $\Delta E>0$,则表示 $E_A>E_B$;若 $\Delta E<0$,则表示 $E_A<E_B$。

当 $E_A\neq E_B$ 时,要通过尾数的移位来改变 E_A 或 E_B,使 $E_A=E_B$ 相等。对阶的规则是:小

阶向大阶看齐。即阶码小的数的尾数右移,每右移一位,阶码加1,直到两数的阶码相等为止。例如:

若 $E_A=E_B$,则无须对阶。

若 $E_A>E_B$,则 M_B 右移。每右移一位,$E_B+1\to E_B$,直至 $E_A=E_B$ 为止。

若 $E_A<E_B$,则 M_A 右移。每右移一位,$E_A+1\to E_A$,直至 $E_A=E_B$ 为止。

尾数右移后,应对尾数进行舍入。

(2) 尾数加/减

对阶之后,就可以进行尾数加/减,即:

$$M_A\pm M_B\to M_C$$

其算法与前面介绍的定点加/减法相同。

(3) 尾数结果规格化

尾数加/减运算之后得到的数可能不是规格化数,为了增加有效数字的位数,提高运算精度,必须进行结果规格化操作。规格化的尾数 M 应该满足下列条件:

$$\frac{1}{2}\leqslant|M|<1$$

设尾数用双符号位补码表示,经过加/减运算之后,可能出现以下6种情况:

① 00.1 ××…×

② 11.0 ××…×

③ 00.0 ××…×

④ 11.1 ××…×

⑤ 01.×××…×

⑥ 10.×××…×

第①种和第②种情况,符合规格化数的定义,已是规格化数。

第③种和第④种情况不是规格化数,需要使尾数左移以实现规格化,这个过程称为左规。尾数每左移一位,阶码相应减1($E_C-1\to E_C$),直至成为规格化数为止。只要满足下列条件:

$$左规=\overline{C}_{s1}\overline{C}_{s2}\overline{C}_1+C_{s1}C_{s2}C_1$$

就进行左规,左规可以进行多次。式中,C_{s1}、C_{s2} 表示尾数 M_C 的两个符号位,C_1 为 M_C 的最高数值位。

第⑤种和第⑥种情况在定点加减运算中称为溢出,但在浮点加减运算中只表明此时尾数的绝对值大于1,而并非真正的溢出。这种情况应将尾数右移以实现规格化。这个过程称为右规。尾数每右移一位,阶码相应加1($E_C+1\to E_C$)。右规的条件如下:

$$右规=C_{s1}\oplus C_{s2}$$

右规最多只有一次。

(4) 舍入

由于受到硬件的限制,在对阶和右规处理之后有可能将尾数的低位丢失,这会引起一些误差。舍入方法有很多种,最简单的是恒舍法,即无条件的丢掉正常尾数最低位之后的全部数值。

(5) 溢出判断

与定点加减法一样,浮点加减运算最后一步也需进行溢出判断。在前面已经指出,当尾数之和(差)出现 10.×××…×或 01.×××…×时,并不表示溢出,只有将此数右规后,再根据阶码来判断浮点运算结果是否溢出。

浮点数的溢出情况由阶码的符号决定,若阶码也用双符号位补码表示,

当$[E_C]_{补}=01,\times\times\times\cdots\times$,表示上溢。此时,浮点数真正溢出,机器需停止运算,做溢出中断处理。

当$[E_C]_{补}=10,\times\times\times\cdots\times$,表示下溢。浮点数值趋于零,机器不做溢出处理,而是按机器零处理。

2. 浮点数加减运算举例

有两浮点数为

$$A=0.101110\times2^{-01}$$

$$B=-(0.101011)\times2^{-10}$$

尾数和阶码均为二进制表示,假设这两数的格式为:阶码 4 位,用移码(偏置值为 2^3)表示;尾数 8 位,用补码表示,包含一位符号位,即:

阶码　　尾数

$$[A]_{浮}=0111;0.1011100$$

$$[B]_{浮}=0110;1.0101010$$

(1) 对阶

求阶差:　　$\Delta E = E_A-E_B=-1-(-2)=1$

$\Delta E=1$,表示 $E_A>E_B$。按对阶规则,将 M_B 右移一位,$E_B+1\rightarrow E_B$,得:

$$[B]'_{浮}=0111;1.1010101$$

(2) 尾数求和

$$\begin{array}{r} 00.1011100 \\ +\ 11.1010101 \\ \hline 00.0110001 \end{array}$$

(3) 尾数结果规格化

由于结果的尾数是非规格化的数,故应左规。尾数左移一位,阶码减 1。最后结果是:

$$[A+B]_{浮}=0110;0.1100010$$

即

$$A+B=(0.110001)\times2^{-10}$$

(4) 舍入及溢出判断

运算结果不需要舍入处理,且阶码未发生溢出。

4.6.2 浮点乘除运算

设两个非 0 的规格化浮点数分别为:

$$A=M_A\times2^{E_A}$$

$$B=M_B\times2^{E_B}$$

规格化浮点数 A、B 乘除运算通式为：

$$(M_A, E_A) \times (M_B, E_B) = (M_A \times M_B, E_A + E_B)$$
$$(M_A, E_A) \div (M_B, E_B) = (M_A \div M_B, E_A - E_B)$$

1. 乘法步骤

两浮点数相乘，其乘积的阶码应为相乘两数的阶码之和，其乘积的尾数应为相乘两数的尾数之积。即：

$$A \times B = (M_A \times M_B) \times 2^{(E_A + E_B)}$$

(1) 阶码相加

如果阶码用补码表示，阶码相加之后无须校正；当阶码用偏置值为 2^n 的移码表示时，阶码相加后要减去一个偏置值 2^n。

因为 $[E_A]_{移} = 2^n + E_A$，$[E_B]_{移} = 2^n + E_B$，$[E_A + E_B]_{移} = 2^n + (E_A + E_B)$

而 $[E_A]_{移} + [E_B]_{移} = 2^n + E_A + 2^n + E_B = 2^n + (E_A + E_B) + 2^n$

所以 $[E_A + E_B]_{移} = [E_A]_{移} + [E_B]_{移} - 2^n$

显然，此时阶码和中多了一个偏置值 2^n，应将它减去。另外，阶码相加后有可能产生溢出，此时应另作处理。

(2) 尾数相乘

若 M_A、M_B 都不为 0，则可进行尾数乘法。尾数乘法的算法与前述定点数乘法算法相同。

(3) 尾数结果规格化

由于 A、B 均是规格化数，所以尾数相乘后的结果一定落在下列范围内：

$$\frac{1}{4} \leqslant |M_A \times M_B| < 1$$

当 $\frac{1}{2} \leqslant |M_A \times M_B| < 1$ 时，乘积已是规格化数，无须再进行规格化操作；当 $\frac{1}{4} \leqslant |M_A \times M_B| < \frac{1}{2}$ 时，则需要左规一次。左规时调整阶码后如果发生阶码下溢，则作机器零处理。

2. 除法步骤

两浮点数相除，其商的阶码应为相除两数的阶码之差，其商的尾数应为相除两数的尾数之商。即：

$$A \div B = (M_A \div M_B) \times 2^{(E_A - E_B)}$$

(1) 尾数调整

为了保证商的尾数是一个定点小数，首先需要检测 $|M_A| < |M_B|$。如果不小于，则 M_A 右移一位，$E_A + 1 \rightarrow E_A$，称为尾数调整。因为 A、B 都是规格化数，所以最多调整一次。

(2) 阶码相减

如果阶码用补码表示，阶码相减之后无须校正；当阶码用偏置值为 2^n 的移码表示时，阶码相减后要加上一个偏置值 2^n。阶码相减后，如有溢出，应另作处理。

(3) 尾数相除

若 M_A、M_B 都不为0,则可进行尾数除法。尾数除法的算法与前述定点数除法算法相同。因为开始时已进行了尾数调整,所以运算结果一定落在规格化范围内,即:

$$\frac{1}{2}\leqslant|M_A\div M_B|<1$$

4.6.3 浮点运算器的实现

由于浮点运算分成阶码和尾数两部分,因此浮点运算器的实现比定点运算器复杂得多。分析上述的浮点四则运算可以发现,对于阶码只有加、减运算,对于尾数则有加、减、乘、除4种运算。可见浮点运算器主要由两个定点运算部件组成,一个是阶码运算部件,用来完成阶码加、减,以及控制对阶时小阶的尾数右移次数和规格化时对阶码的调整;另一个是尾数运算部件,用来完成尾数的四则运算以及判断尾数是否已规格化。此外,还需要有溢出判断电路等。

现代计算机可把浮点运算部件做成任选件,或称为协处理器。所谓协处理器,是因为它只能协助主处理器工作,不能单独工作。

4.7 十进制整数的加法运算

一些通用计算机中设有十进制数据表示,可以直接对十进制整数进行算术运算。下面介绍十进制整数的加法运算和十进制加法器。

4.7.1 一位十进制加法运算

在计算机中,十进制数是用BCD码表示的,BCD码由4位二进制数表示,按二进制加法规则进行加法。十进制数的进位是10,而4位二进制数的进位是16,为此需要进行必要的十进制校正,才能使该进位正确。因为不同的BCD码对应的十进制校正规律是不一样的,所以硬件实现也是不同的。

1. 8421码加法运算

8421码的加法规则如下:

① 两个十进制数的8421码相加时,按"逢二进一"的原则进行。

② 当和≤9,无须校正。

③ 当和>9,则+6校正。

④ 在做+6校正的同时,将产生向上一位的进位。

8421码的校正关系如表4-9所示。

表4-9　8421码的校正关系

十进制数	8421码 C_4 S_4 S_3 S_2 S_1	校正前的二进制数 C_4' S_4' S_3' S_2' S_1'	校正关系
0～9	0 0 0 0 0 ⋮ 0 1 0 0 1	0 0 0 0 0 ⋮ 0 1 0 0 1	不校正

续表

十进制数	8421码 C_4 S_4 S_3 S_2 S_1	校正前的二进制数 C_4' S_4' S_3' S_2' S_1'	校正关系
10	1 0 0 0 0	0 1 0 1 0	+6校正
11	1 0 0 0 1	0 1 0 1 1	
12	1 0 0 1 0	0 1 1 0 0	
13	1 0 0 1 1	0 1 1 0 1	
14	1 0 1 0 0	0 1 1 1 0	
15	1 0 1 0 1	0 1 1 1 1	
16	1 0 1 1 0	1 0 0 0 0	
17	1 0 1 1 1	1 0 0 0 1	
18	1 1 0 0 0	1 0 0 1 0	
19	1 1 0 0 1	1 0 0 1 1	

根据校正关系，很容易得到：

$$校正函数=C_4'+S_4'S_3'+S_4'S_2'$$

向上一位的进位 C_4＝校正函数。

2. 余3码加法运算

余3码的加法规则如下：

① 两个十进制数的余3码相加，按“逢二进一”的原则进行。

② 若其和没有进位，则减3(即+1101)校正。

③ 若其和有进位，则加3(即+0011)校正。

余3码的校正关系如表4-10所示。

根据校正关系，很容易得到校正函数：$C_4'=0$，−3校正；$C_4'=1$，+3校正。向上一位的进位 $C_4=C_4'$。

表4-10　余3码的校正关系

十进制数	余3码 C_4 S_4 S_3 S_2 S_1	校正前的二进制数 C_4' S_4' S_3' S_2' S_1'	校正关系
0	0 0 0 1 1	0 0 1 1 0	−3校正
1	0 0 1 0 0	0 0 1 1 1	
2	0 0 1 0 1	0 1 0 0 0	
3	0 0 1 1 0	0 1 0 0 1	
4	0 0 1 1 1	0 1 0 1 0	
5	0 1 0 0 0	0 1 0 1 1	
6	0 1 0 0 1	0 1 1 0 0	
7	0 1 0 1 0	0 1 1 0 1	
8	0 1 0 1 1	0 1 1 1 0	
9	0 1 1 0 0	0 1 1 1 1	

续表

十进制数	余 3 码 C_4 S_4 S_3 S_2 S_1	校正前的二进制数 C_4' S_4' S_3' S_2' S_1'	校正关系
10	1 0 0 1 1	1 0 0 0 0	+3 校正
11	1 0 1 0 0	1 0 0 0 1	
12	1 0 1 0 1	1 0 0 1 0	
13	1 0 1 1 0	1 0 0 1 1	
14	1 0 1 1 1	1 0 1 0 0	
15	1 1 0 0 0	1 0 1 0 1	
16	1 1 0 0 1	1 0 1 1 0	
17	1 1 0 1 0	1 0 1 1 1	
18	1 1 0 1 1	1 1 0 0 0	
19	1 1 1 0 0	1 1 0 0 1	

4.7.2 十进制加法器

1. 一位 8421 码加法器

按照校正函数构成的 8421 码加法器如图 4-21 所示。图中上部 4 个全加器(FA)实现二进制求和运算,下部一个全加器和两个半加器(HA)则用来实现+6(+0110)的校正操作。

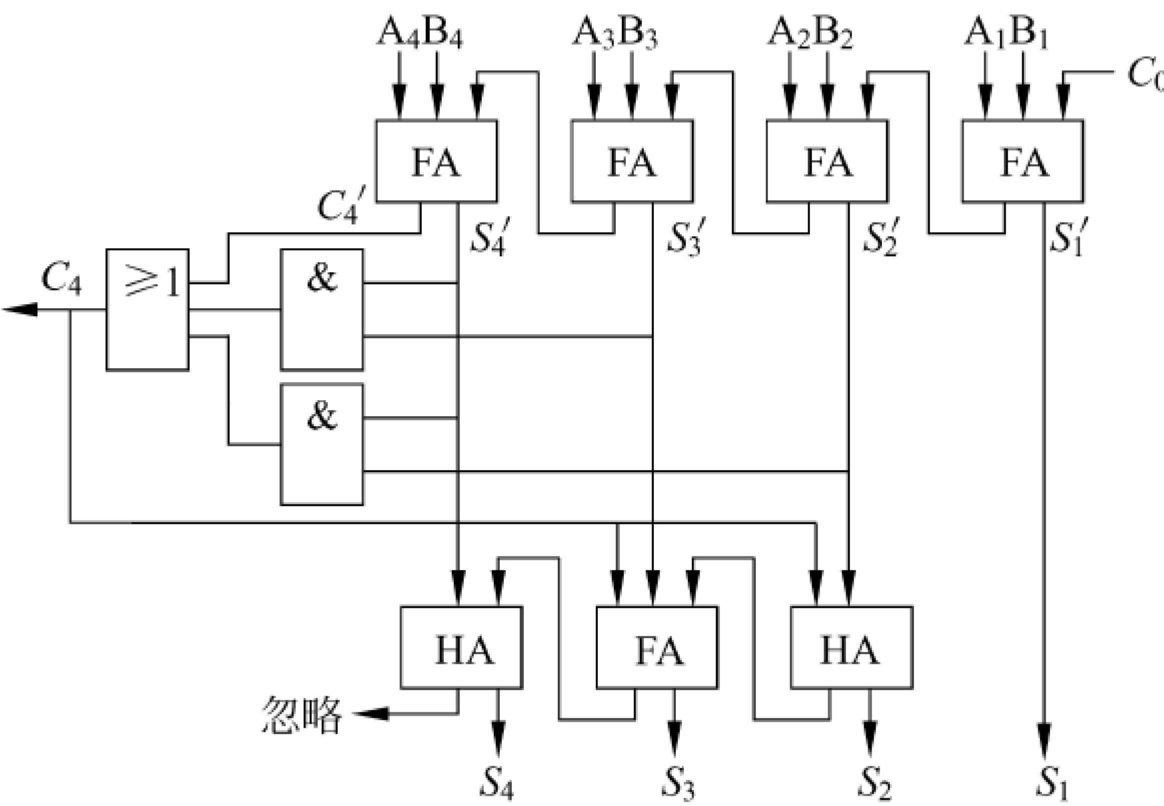


图 4-21 一位 8421 码加法器

2. 一位余 3 码加法器

按照校正函数构成的余 3 码加法器如图 4-22 所示,图中上部 4 个全加器实现二进制求和运算。从表 4-10 可以看出校正前后最低位的值永远是相反的,所以用一个非门使 S_1'求反,这个非门与下部 3 个全加器一起共同实现+3(+0011)或−3(+1101)的校正操作。

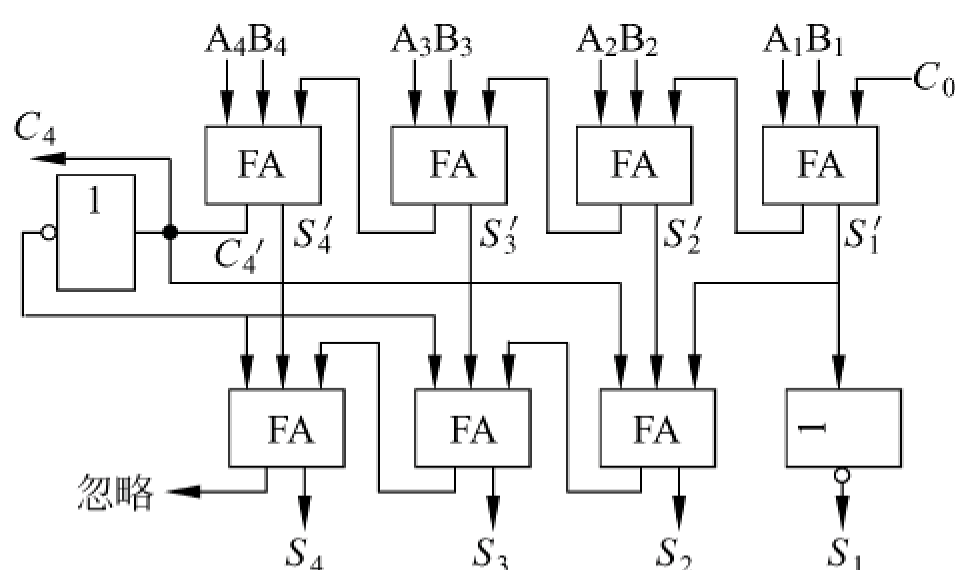


图 4-22　一位余 3 码加法器

4.7.3　多位十进制加法

前面介绍了一位十进制加法规则和加法器，对于多位十进制数来说，完全遵照一位十进制数的加法规则。

例 4-14　用 8421 码求 48＋36。

```
    0100   1000     --- 48
+   0011   0110     --- 36
------------------
    0111   1110     低位和大于 9
+      1←0110       加 6校正，并产生向上一位的进位
------------------
    1000   0100     --- 84
```

所以 48＋36＝84。

例 4-15　用余 3 码求 2631＋1591。

```
    0101   1001   0110   0100     --- 2631
+   0100   1000   1100   0100     --- 1591
---------------------------------
    1010   0010   0010   1000     有进位 +3,无进位 −3
+   1101   0011   0011   1101
---------------------------------
    0111   0101   0101   0101     --- 4222
```

所以 2631＋1591＝4222。

在例 4-15 中，个位和千位在求和时不产生进位，做减 3 校正；十位和百位在求和时产生进位，做加 3 校正。

注意：校正前的余 3 码向高位的进位均有效，校正后的各余 3 码向高位的进位均自动丢失。

对于多位十进制数加法可采用多个 BCD 码加法器，每个 BCD 码加法器就是前述的一个一位十进制加法器，可执行两个一位 BCD 数的加法。若 n 位 BCD 数相加，由从低位至高位采用行波式串行进位的 n 位十进制加法器完成，如图 4-23 所示。

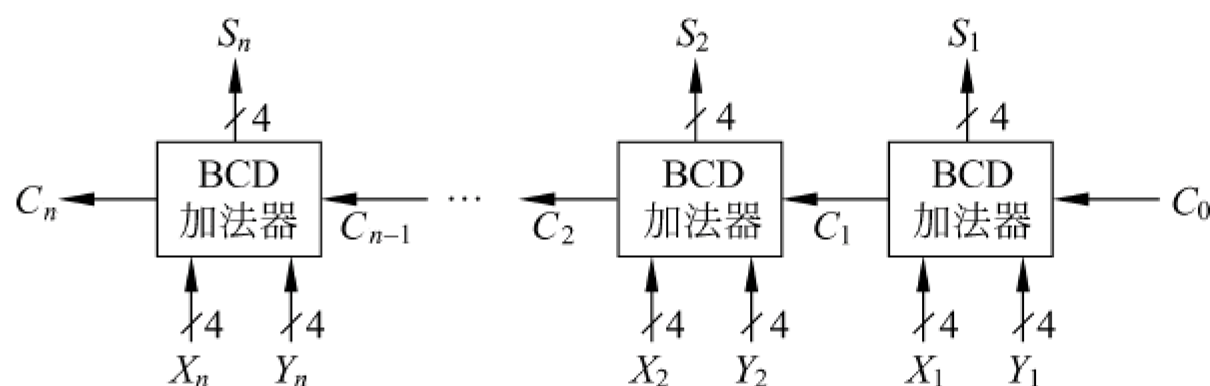


图 4-23　n 位串行十进制加法器

4.8 逻辑运算与实现

计算机在解题过程中,除了要做大量的算术运算外,还需做许多逻辑操作,例如与、或、非、异或等。逻辑运算比算术运算要简单得多,这是因为逻辑运算是按位进行的,位与位之间没有进位与借位的关系。

1. 逻辑非

逻辑非又称求反操作,它对某个寄存器或主存单元中各位代码按位取反。

假设: $X=X_0X_1\cdots X_n$, $Z=Z_0Z_1\cdots Z_n$

则: $Z_i=\overline{X}_i \quad (i=0,1,\cdots,n)$

逻辑非可利用非门(反相器)实现。

2. 逻辑乘

逻辑乘就是将两个寄存器或主存单元中的每一相应位的代码进行按位与操作。

假设: $X=X_0X_1\cdots X_n$, $Y=Y_0Y_1\cdots Y_n$, $Z=Z_0Z_1\cdots Z_n$

则: $Z_i=X_i\wedge Y_i \quad (i=0,1,\cdots,n)$

一位二进制数的逻辑乘规则如表 4-11 所示。

表 4-11 二进制逻辑乘规则

$X_i\wedge Y_i$	Z_i	$X_i\wedge Y_i$	Z_i
0 0	0	1 0	0
0 1	0	1 1	1

逻辑乘可以用与门来实现,也可以用或门和非门实现,即:$Z_i=X_i\wedge Y_i=\overline{\overline{X}_i\vee\overline{Y}_i}$。

3. 逻辑加

逻辑加就是将两个寄存器或主存单元中的每一相应位的代码进行按位或操作。

假设: $X=X_0X_1\cdots X_n$, $Y=Y_0Y_1\cdots Y_n$, $Z=Z_0Z_1\cdots Z_n$

则: $Z_i=X_i\vee Y_i \quad (i=0,1,\cdots,n)$

一位二进制数的逻辑加规则如表 4-12 所示。

表 4-12 二进制逻辑加规则

$X_i\vee Y_i$	Z_i	$X_i\vee Y_i$	Z_i
0 0	0	1 0	1
0 1	1	1 1	1

逻辑加可以用或门来实现,也可以用与门和非门实现,即 $Z_i=X_i\vee Y_i=\overline{\overline{X}_i\wedge\overline{Y}_i}$。

4. 逻辑异或

逻辑异或又称按位加,它对两个寄存器或主存单元中各位的代码求模 2 和。

假设：$X=X_0X_1\cdots X_n$，$Y=Y_0Y_1\cdots Y_n$，$Z=Z_0Z_1\cdots Z_n$

则：$Z_i=X_i \oplus Y_i \quad (i=0,1,\cdots,n)$

一位二进制数的按位加的运算规则如表 4-13 所示。

表 4-13　二进制按位加规则

$X_i \oplus Y_i$	Z_i	$X_i \oplus Y_i$	Z_i
0　0	0	1　0	1
0　1	1	1　1	0

按位加采用异或门实现。

逻辑运算操作既可以由各种专门设置的电路来实现，也可以利用算术逻辑运算部件(ALU)来实现，但在进行逻辑运算时要封锁进位链。

4.9　运算器的基本组成与实例

运算器是在控制器的控制下实现其功能的。运算器不仅可以完成数据信息的算术逻辑运算，还可以作为数据信息的传送通路。

4.9.1　运算器结构

1. 运算器的基本组成

基本的运算器包含以下几个部分：实现基本算术、逻辑运算功能的 ALU，提供操作数与暂存结果的寄存器组，有关的判别逻辑和控制电路等。将这些功能模块连接成一个整体时，需要解决一个问题，就是如何向 ALU 提供操作数？一种方法是在 ALU 输入端加多路选择器，另一种方法是在 ALU 输入端加一级锁存器(暂存器)。

运算器内的各功能模块之间的连接也广泛采用总线结构，这个总线称为运算器的内部总线，ALU 和各寄存器都挂在上面。应当引起大家注意的是，这里所说的总线与第 1 章中提到的系统总线的含义不同，运算器的内部总线是 CPU 的内部数据通路，因此只有数据线。

(1) 带多路选择器的运算器

图 4-24 为带多路选择器的运算器，各寄存器可以独立、多路地将数据送至 ALU 的多路选择器，使 ALU 有选择地同时获得两路输入数据。运算器的内部总线是一组单向传送的数据线，它将运算结果送往各寄存器，由寄存器的同步打入脉冲 CP_i 将内部总线上的数据送入 R_i。如果同时发出几个打入脉冲，则可将总线上的同一数据同时送入几个相关的寄存器中。

(2) 带输入锁存器的运算器

图 4-25 为带输入锁存器的运算器，运算器的内部总线是一组双向传送的数据线。为了进行双操作数之间的运算操作，ALU 输入端前设置了一级锁存器，可暂存操作数。例如，要实现$(R_0)+(R_1)\rightarrow R_2$，可通过内部总线先将 R_0 中的数据送入锁存器 1，再通过内部总线将 R_1 中的数据送入锁存器 2，然后相加，并将结果经总线送入 R_2。

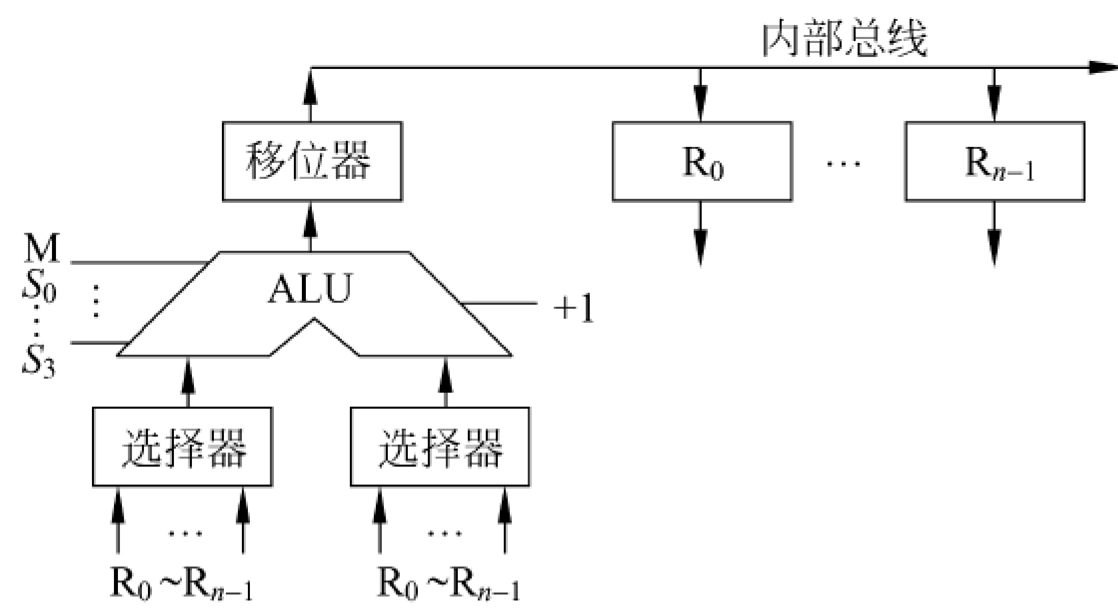


图 4-24　带多路选择器的运算器

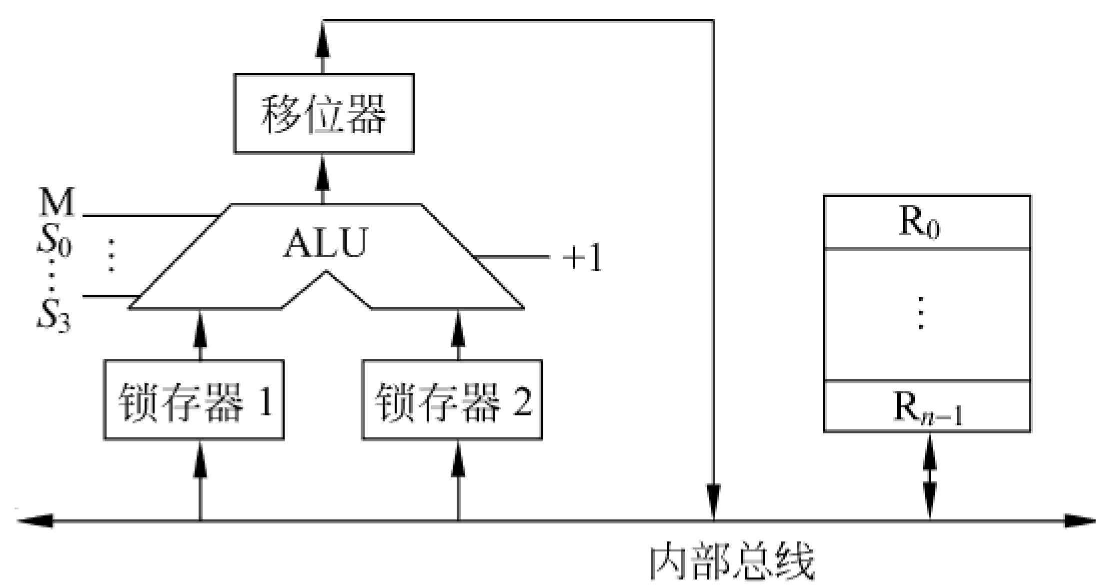


图 4-25　带输入锁存器的运算器

2. 运算器的内部总线结构

运算器的内部总线大体有以下 3 种结构形式。

(1) 单总线结构运算器

图 4-25 就是单总线结构运算器。这种结构的运算器实现一次双操作数的运算需要分成 3 步,它的主要缺点是操作速度慢。

(2) 双总线结构运算器

图 4-26(a)为双总线结构运算器。两个操作数可以分别通过总线 1 和总线 2 同时送到 ALU 去进行运算,并且立即可以得到运算的结果。但是,ALU 的输出不能直接送到总线上去,这是因为此时两条总线都被操作数所占据着,所以必须在 ALU 的输出端设置一个缓冲器,先将运算结果送入缓冲器,下一步再把结果送至目的寄存器。显然,它的执行速度比单总线要快,每次操作比单总线少一步。

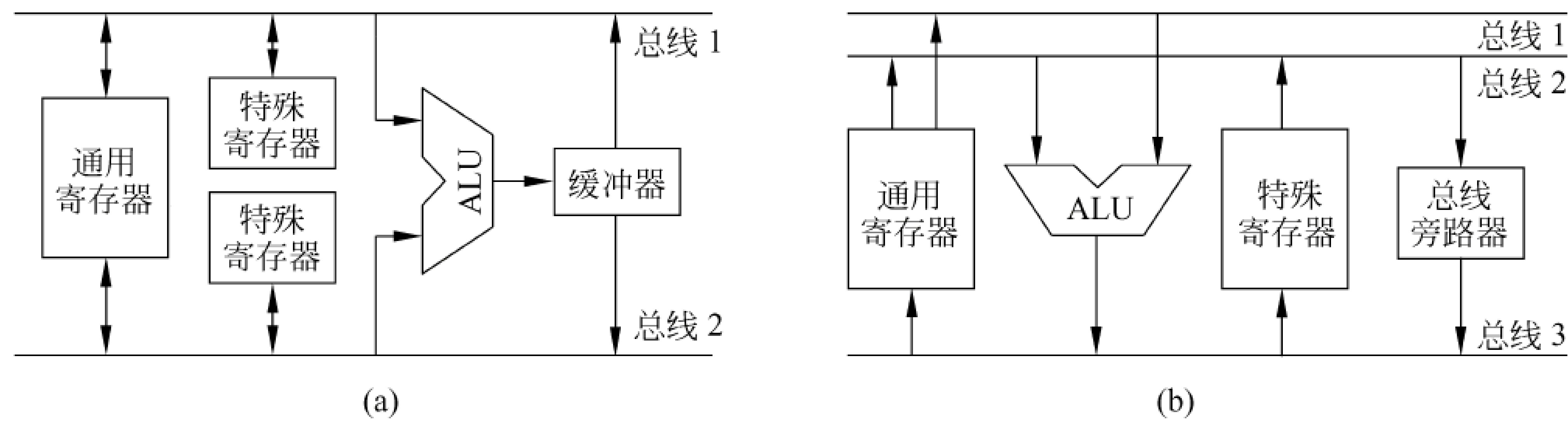


图 4-26　多总线结构运算器内的数据通路

(3) 三总线结构运算器

三总线结构运算器如图 4-26(b)所示。ALU 的两个输入端分别由两条总线供给,输出与第三条总线相连,这样算术和逻辑运算操作就可以在一步控制之内完成。如果某一个数不需要运算和修改,而需要直接由总线 2 传到总线 3,可通过总线旁路器把数据送出,而不必借助于 ALU。三总线结构的特点是操作速度快,但控制较前两种复杂。

4.9.2 ALU 举例

1. ALU 电路

ALU 即算术逻辑单元,它是既能完成算术运算又能完成逻辑运算的部件。由于无论是加、减、乘、除运算,最终都能归结为加法运算。因此,ALU 的核心首先应当是一个并行加法器,同时也能执行像“与”“或”“非”“异或”这样的逻辑运算。由于 ALU 能完成多种功能,所以 ALU 又称多功能函数发生器。

2. 4 位 ALU 芯片

过去大多数 ALU 是 4 位的,目前随着集成电路技术的发展,多位的 ALU 已相继问世。为了说明原理,仍以典型的 4 位 ALU 芯片(74181)为例介绍 ALU 的结构及应用。

74181 能执行 16 种算术运算和 16 种逻辑运算。工作于正逻辑或负逻辑的 74181 的框图分别如图 4-27(a)、(b)所示。以负逻辑为例,其中 $\overline{A}_3 \sim \overline{A}_0$ 和 $\overline{B}_3 \sim \overline{B}_0$ 是两个操作数,$\overline{F}_3 \sim \overline{F}_0$ 为输出结果;C_n 表示最低位的外来进位,C_{n+4} 是向高位的进位;$\overline{G}$ 为组进位产生函数输出,$\overline{P}$ 为组进位传递函数输出;M 表示工作方式(M=0 为算术操作,M=1 为逻辑操作),$S_3 \sim S_0$ 为功能选择线。

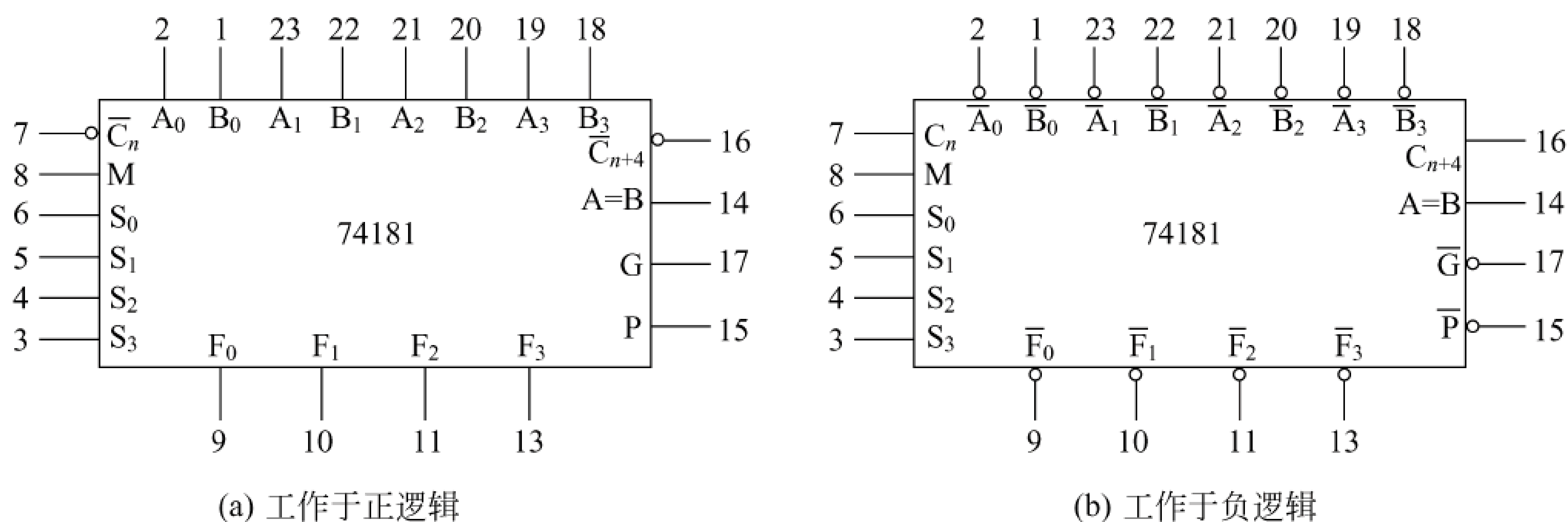


图 4-27 74181 芯片

表 4-14 给出了 74181 的算术/逻辑运算功能。

在表 4-14 中,“+”表示逻辑或运算,“加”表示算术加运算。M 的值用来区别算术还是逻辑运算,$S_3 \sim S_0$ 的不同取值可实现不同的操作。例如,以负逻辑为例,当 M=1,$S_3 \sim S_0$=1001 时,做逻辑运算 $A \oplus B$;当 M=0,$S_3 \sim S_0$=1001 时,做算术运算,此时若 C_n=0,完成 A 加 B,若 C_n=1,完成 A 加 B 加 1。

表 4-14　74181 的算术/逻辑运算功能表

工作选择 $S_3S_2S_1S_0$	负逻辑			正逻辑		
	逻辑运算（M=1）	算术运算（M=0）$C_n=0$（无进位）	算术运算（M=0）$C_n=1$（有进位）	逻辑运算（M=1）	算术运算（M=0）$\bar{C}_n=1$（无进位）	算术运算（M=0）$\bar{C}_n=0$（有进位）
0000	$F=\bar{A}$	$F=A$ 减 1	$F=A$	$F=\bar{A}$	$F=A$	$F=A$ 加 1
0001	$F=\overline{AB}$	$F=AB$ 减 1	$F=AB$	$F=\overline{A+B}$	$F=A+B$	$F=(A+B)$加 1
0010	$F=\bar{A}+B$	$F=A\bar{B}$ 减 1	$F=A\bar{B}$	$F=\bar{A}B$	$F=A+\bar{B}$	$F=(A+\bar{B})$加 1
0011	$F=1$	$F=$减 1	$F=0$	$F=0$	$F=$减 1	$F=0$
0100	$F=\overline{A+B}$	$F=A$ 加$(A+\bar{B})$	$F=A$ 加$(A+\bar{B})$加 1	$F=\overline{AB}$	$F=A$ 加 $A\bar{B}$	$F=A$ 加 $A\bar{B}$ 加 1
0101	$F=\bar{B}$	$F=AB$ 加$(A+\bar{B})$	$F=AB$ 加$(A+\bar{B})$加 1	$F=\bar{B}$	$F=(A+B)$加 $A\bar{B}$	$F=(A+B)$加 $A\bar{B}$ 加 1
0110	$F=\overline{A\oplus B}$	$F=A$ 减 B 减 1	$F=A$ 减 B	$F=A\oplus B$	$F=A$ 减 B 减 1	$F=A$ 减 B
0111	$F=A+\bar{B}$	$F=A+\bar{B}$	$F=(A+\bar{B})$加 1	$F=A\bar{B}$	$F=A\bar{B}$ 减 1	$F=A\bar{B}$
1000	$F=\bar{A}B$	$F=A$ 加$(A+B)$	$F=A$ 加$(A+B)$加 1	$F=\bar{A}+B$	$F=A$ 加 AB	$F=A$ 加 AB 加 1
1001	$F=A\oplus B$	$F=A$ 加 B	$F=A$ 加 B 加 1	$F=\overline{A\oplus B}$	$F=A$ 加 B	$F=A$ 加 B 加 1
1010	$F=B$	$F=A\bar{B}$ 加$(A+B)$	$F=A\bar{B}$ 加$(A+B)$加 1	$F=B$	$F=(A+\bar{B})$加 AB	$F=(A+\bar{B})$加 AB 加 1
1011	$F=A+B$	$F=A+B$	$F=(A+B)$加 1	$F=AB$	$F=AB$ 减 1	$F=AB$
1100	$F=0$	$F=A$ 加 A *	$F=A$ 加 A 加 1	$F=1$	$F=A$ 加 A *	$F=A$ 加 A 加 1
1101	$F=A\bar{B}$	$F=AB$ 加 A	$F=AB$ 加 A 加 1	$F=A+\bar{B}$	$F=(A+B)$加 A	$F=(A+B)$加 A 加 1
1110	$F=AB$	$F=A\bar{B}$ 加 A	$F=A\bar{B}$ 加 A 加 1	$F=A+B$	$F=(A+\bar{B})$加 A	$F=(A+\bar{B})$加 A 加 1
1111	$F=A$	$F=A$	$F=A$ 加 1	$F=A$	$F=A$ 减 1	$F=A$

注：* 表示 A 加 $A=2A$，算术左移一位。

3. ALU 的应用

74181 的 4 位作为一个小组，小组间既可以采用串行进位，也可以采用并行进位。当采用串行进位时，只要把低一片的 C_{n+4} 与高一片的 C_n 相连即可。当采用组间并行进位时，需要增加一片 74182，这是一个先行进位部件。74182 芯片方框图如图 4-28 所示。74182 可以产生 3 个进位信号 C_{n+x}、C_{n+y}、C_{n+z}，并且还产生大组进位产生函数 $\overline{G}$ 和进位传递函数 $\overline{P}$。

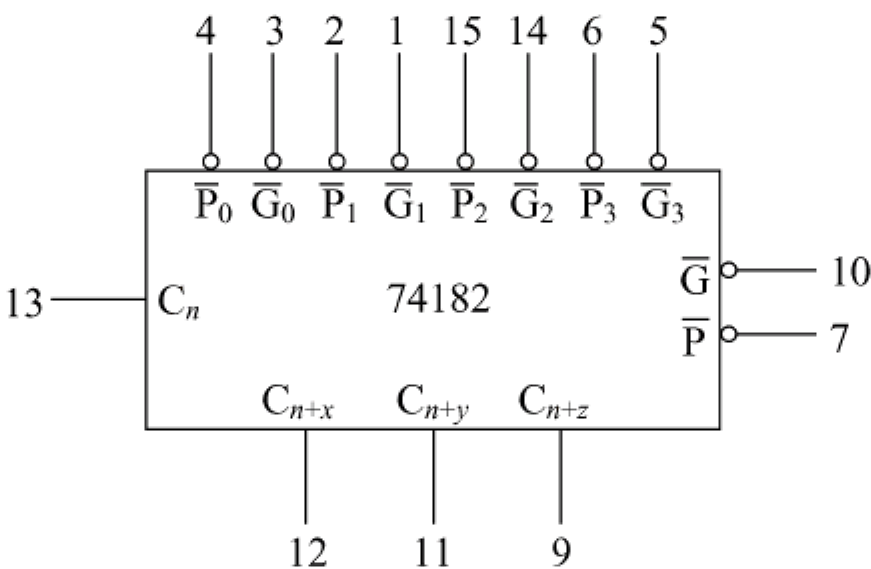


图 4-28 74182 芯片

图 4-29 是由 8 片 74181 和 2 片 74182 构成的 32 位两级行波 ALU。各片 74181 输出的组进位产生函数 $\overline{G}_i$ 和组进位传递函数 $\overline{P}_i$ 作为 74182 的输入，而 74182 输出的进位信号 C_{n+x}、C_{n+y}、C_{n+z} 作为 74181 的输入，74182 输出的大组进位产生函数 $\overline{G}$ 和大组进位传递函数 $\overline{P}$ 可作为更高一级 74182 的输入。

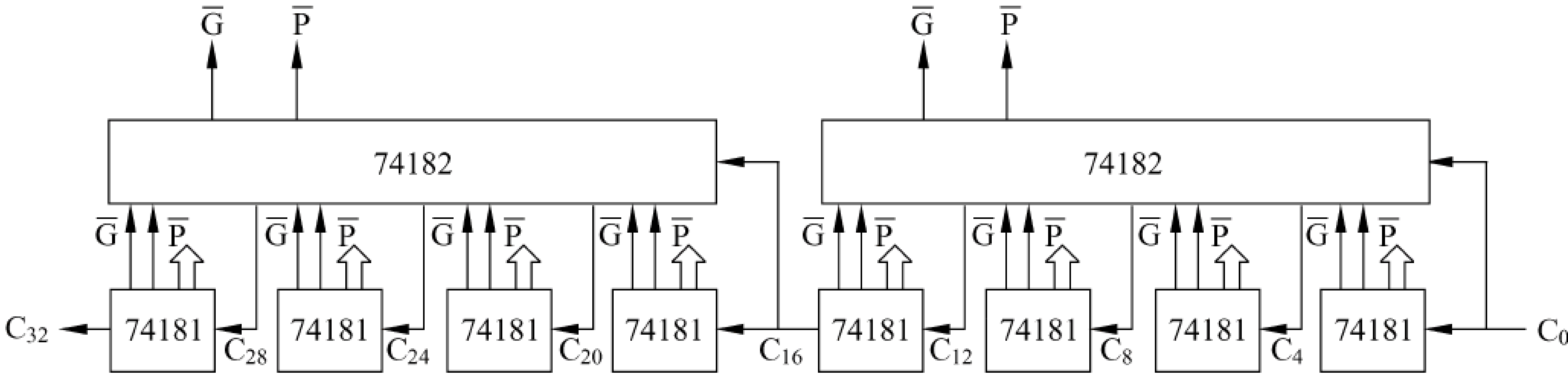


图 4-29 32 位两级行波 ALU

74181 的结构适合于组成各种位数的 ALU 部件。表 4-15 列出了具有不同位数和结构的 ALU 所需的芯片数目和加法时间。

4.9.3 浮点运算器举例

目前在微机系统中往往配置有专门的浮点运算部件，例如 PC 系列机中的 80x87 就是浮点协处理器。对于 486SX 以下的微机，80x87 是任选件；而对于 486DX 及其以上的微机，80x87 已被集成在 CPU 芯片中。

表 4-15 具有不同位数和结构的 ALU

位数	ALU 结构	总的加法时间(ns)	芯片片数	
			74181	74182
4	一级 ALU	21	1	0
8	一级行波 ALU	36	2	0
16	一级行波 ALU	60	4	0
16	两级 ALU	36	4	1
32	一级行波 ALU	110	8	0

续表

位　　数	ALU 结构	总的加法时间(ns)	芯 片 片 数	
			74181	74182
32	两级行波 ALU	62	8	2
48	一级行波 ALU	160	12	0
48	两级行波 ALU	101	12	3
48	三级 ALU	64	12	4
64	一级行波 ALU	210	16	0
64	两级行波 ALU	136	16	4
64	三级 ALU	88	16	5

1. 80x87 的数据格式

80x87 可处理 7 种不同的数据类型,这些数据类型的格式如图 4-30 所示。对整数来说,最高位为符号位,用补码表示,有 16、32 和 64 位 3 种格式。压缩的十进制数串是用特殊形式表示的整数。十进制数的一位用 4 位二进制表示,80 位的低 72 位表示 18 位十进制数,最高位为符号位,正数和负数都是以原码形式存储的。浮点数有 32、64 和 80 位 3 种格式,阶码的底为 2,用移码表示,尾数用原码表示。

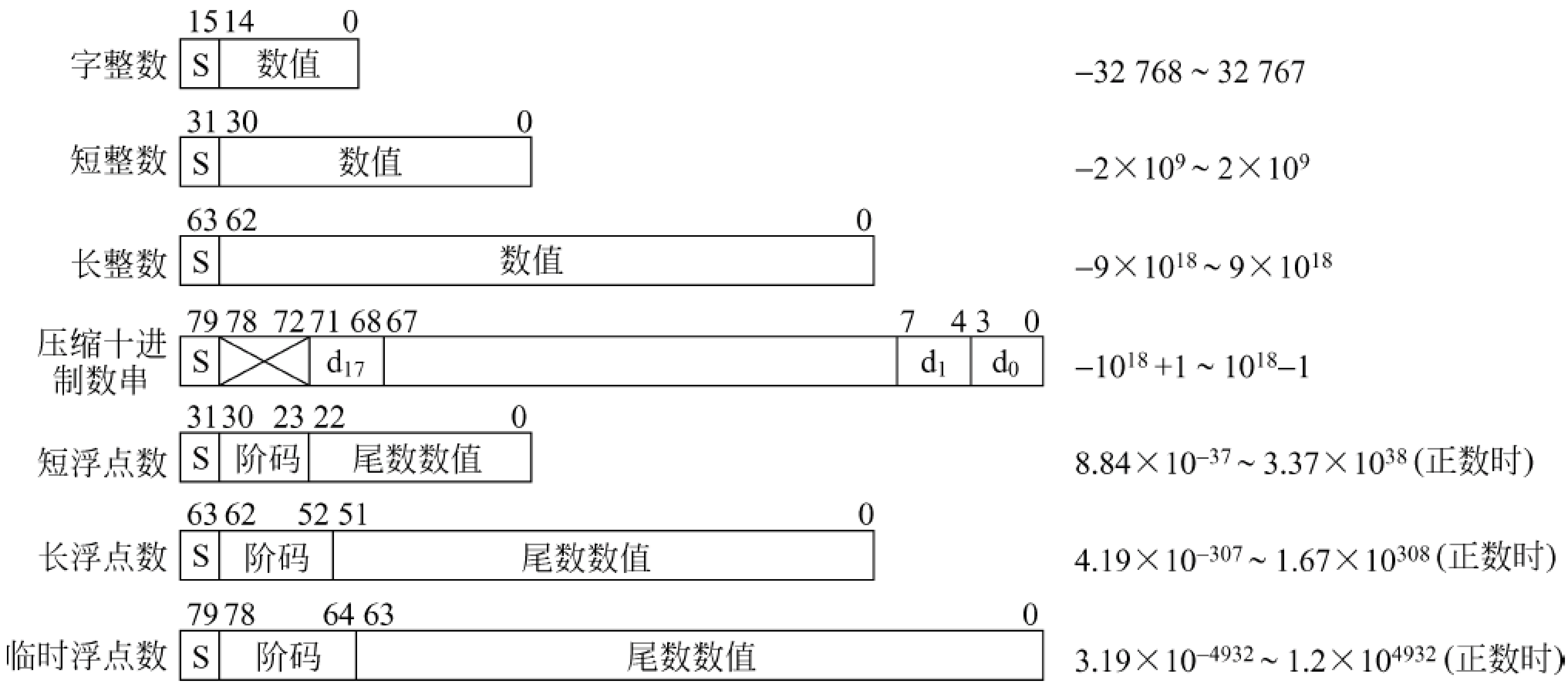


图 4-30　80x87 的数据格式

2. 80x87 的内部结构

图 4-31 为 80x87 的内部结构。它是由总线控制逻辑部件、数据接口与控制部件、浮点运算部件 3 个主要功能模块组成的。

在 80x87 的浮点运算部件中,分别设置了阶码(指数)运算部件与尾数运算部件,并设有加速移位操作的移位器。它们通过指数总线和尾数总线与 8 个 80 位字长的寄存器组相连。

80x87 从主存取数或向主存写数时,均用 80 位的临时浮点数与其他数据类型执行自动转换。在 80x87 中的全部数据都以 80 位临时浮点数的形式表示。

80x87 与主微处理器协同工作,微处理器执行所有的常规指令,而 80x87 只执行专门的

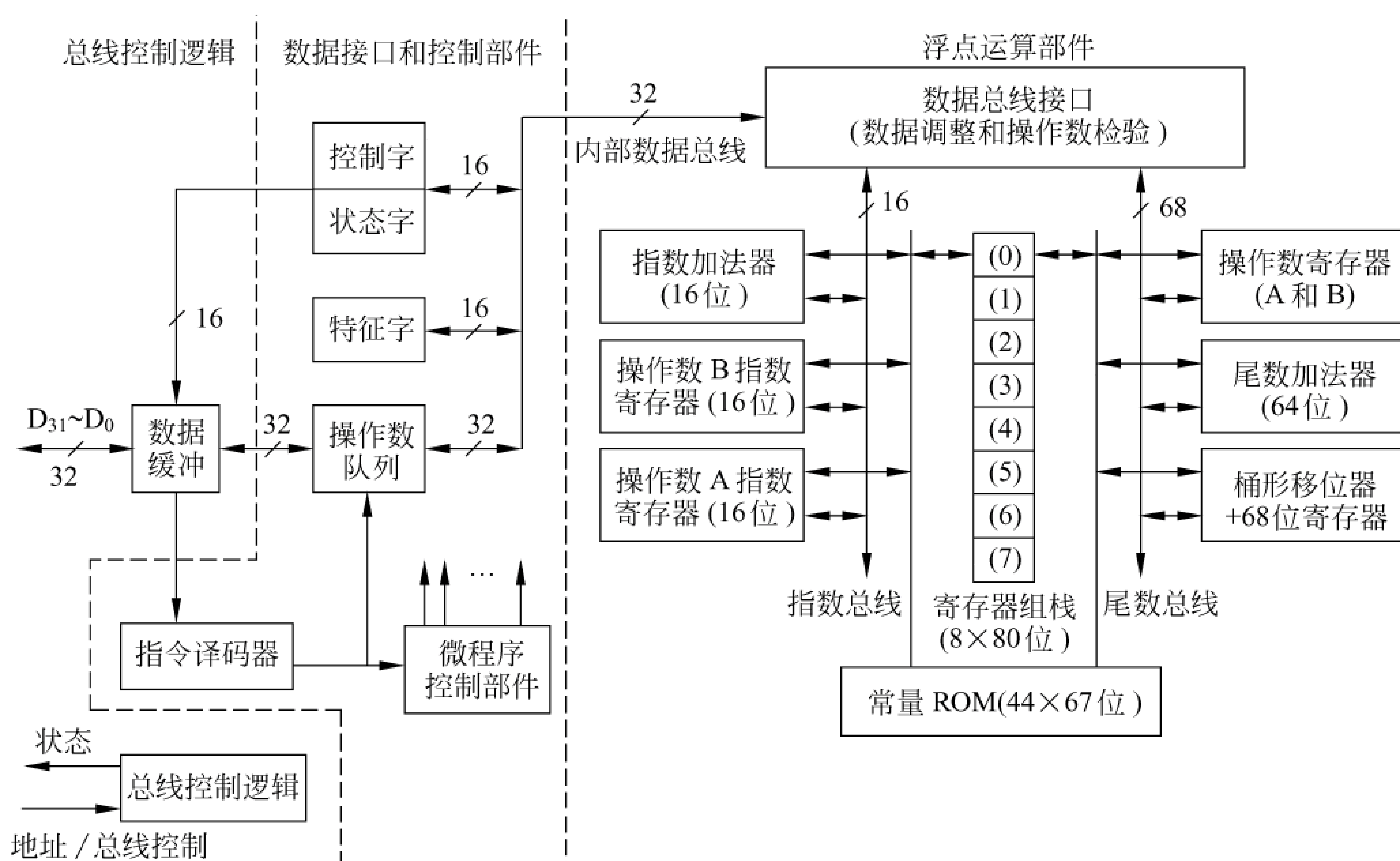


图 4-31　80x87 的内部结构

算术协处理器指令,该指令称为换码(ESC)指令。微处理器和协处理器可以同时或并行执行各自的指令。

习　　题

4-1　证明:在全加器里进位传递函数 $P=A_i+B_i=A_i\oplus B_i$。

4-2　某加法器采用组内并行、组间并行的进位链,4 位一组,写出进位信号 C_6 的逻辑表达式。

4-3　设计一个 9 位先行进位加法器,每 3 位为一组,采用两级先行进位线路。

4-4　已知 X 和 Y,试用它们的变形补码计算出 $X+Y$,并指出结果是否溢出。

(1) $X=0.11011$,　$Y=0.11111$

(2) $X=0.11011$,　$Y=-0.10101$

(3) $X=-0.10110$,　$Y=-0.00001$

(4) $X=-0.11011$,　$Y=0.11110$

4-5　已知 X 和 Y,试用它们的变形补码计算出 $X-Y$,并指出结果是否溢出。

(1) $X=0.11011$,　$Y=-0.11111$

(2) $X=0.10111$,　$Y=0.11011$

(3) $X=0.11011$,　$Y=-0.10011$

(4) $X=-0.10110$,　$Y=-0.00001$

4-6　已知 $X=0.1011$,$Y=-0.0101$。

求 $\left[\frac{1}{2}X\right]_{补}$,$\left[\frac{1}{4}X\right]_{补}$,$[-X]_{补}$,$\left[\frac{1}{2}Y\right]_{补}$,$\left[\frac{1}{4}Y\right]_{补}$,$[-Y]_{补}$。

4-7　设下列数据长 8 位,包括一位符号位,采用补码表示,分别写出每个数据右移或左移两位之后的结果。

(1) 0.1100100

(2) 1.0011001

(3) 1.1100110

(4) 1.0000111

4-8　分别用原码乘法和补码乘法计算 $X\times Y$。

(1) $X=0.11011$，　$Y=-0.11111$

(2) $X=-0.11010$，　$Y=-0.01110$

4-9　根据补码两位乘法规则推导出补码 3 位乘法的规则。

4-10　分别用原码和补码加减交替法计算 $X\div Y$。

(1) $X=0.10101$，　$Y=0.11011$

(2) $X=-0.10101$，　$Y=0.11011$

(3) $X=0.10001$，　$Y=-0.10110$

(4) $X=-0.10110$，　$Y=-0.11011$

4-11　已知:$X=-7.25$,$Y=28.5625$

(1) 将 X、Y 分别转换成二进制浮点数(阶码占 4 位,尾数占 10 位,各包含一位符号位)。

(2) 用变形补码,求 $X-Y=$?

4-12　设浮点数的阶码和尾数部分均用补码表示,按照浮点数的运算规则,计算下列各题(题中数字均为二进制数):

(1) $X=2^{101}\times(-0.100010)$，　$Y=2^{100}\times(-0.111110)$

(2) $X=2^{-101}\times0.101100$，　$Y=2^{-100}\times(-0.101000)$

(3) $X=2^{-011}\times0.101100$，　$Y=2^{-001}\times(-0.111100)$

求 $X+Y$,$X-Y$。

4-13　设浮点数的阶码和尾数部分均用补码表示,按照浮点数的运算规则,计算下列各题:

(1) $X=2^{3}\times\dfrac{13}{16}$，　$Y=2^{4}\times\left(-\dfrac{9}{16}\right)$

求 $X\times Y$。

(2) $X=2^{3}\times\left(-\dfrac{13}{16}\right)$，　$Y=2^{5}\times\left(\dfrac{15}{16}\right)$

求 $X\div Y$。

4-14　用流程图描述浮点除法运算的算法步骤。

4-15　设计一个一位 5421 码加法器。

4-16　某计算机利用二进制的加法器进行 8421 码的十进制运算,采用的方法是:

(1) 对某一操作数预加 6 后,与另一操作数一起进入二进制加法器。

(2) 有进位产生时,直接得到和的 8421 码。

(3) 没有进位时,反减 6 再得到和的 8421 码。

试求+6、-6 的校正逻辑。

4-17　用 74181 和 74182 芯片构成一个 64 位的 ALU,采用多级分组并行进位链(要求速度尽可能快)。

第 5 章 存储系统和结构

存储系统是由几个容量、速度和价格各不相同的存储器构成的系统。设计一个容量大、速度快、成本低的存储系统是计算机发展的一个重要课题。本章重点讨论主存储器的工作原理、组成方式以及运用半导体存储芯片组成主存储器的一般原则和方法,此外还将介绍高速缓冲存储器和虚拟存储器的基本原理。

5.1 存储系统的组成

存储系统和存储器是两个不同的概念。本节首先介绍各种不同用途的存储器,然后介绍它们是如何构成一个存储系统的。

5.1.1 存储器分类

随着计算机系统结构和存储技术的发展,存储器的种类日益繁多,根据不同的特征可对存储器进行分类。

1. 按存储器在计算机系统中的作用分类

(1) 高速缓冲存储器

高速缓冲存储器(Cache)位于主存和 CPU 之间,用来存放正在执行的程序段和数据,以便 CPU 能高速地使用它们。高速缓冲存储器的存取速度可以与 CPU 的速度相匹配,但存储容量较小,价格较高。目前的高档微机通常将它们或它们的一部分制作在 CPU 芯片中。

(2) 主存储器

主存用来存放计算机运行期间所需要的程序和数据,CPU 可直接随机地进行读/写访问。主存具有一定容量,存取速度较高。由于 CPU 要频繁地访问主存,所以主存的性能在很大程度上影响了整个计算机系统的性能。

(3) 辅助存储器

辅助存储器又称外存储器或后援存储器,它用来存放当前暂不参与运行的程序和数据以及一些需要永久性保存的信息。辅存设在主机外部,容量极大且成本很低,但存取速度较低,而且 CPU 不能直接访问它。辅存中的信息必须调入主存之后,CPU 才能使用。

2. 按存取方式分类

(1) 随机存取存储器(Random Access Memory,RAM)

所谓随机存取是指CPU可以对存储器中的内容随机地存取,CPU对任何一个存储单元的写入和读出时间是一样的,即存取时间相同,与其所处的物理位置无关。RAM读/写方便,使用灵活,主要用作主存,也可用作高速缓冲存储器。

(2) 只读存储器(Read Only Memory,ROM)

ROM可以看作RAM的一种特殊形式,其特点是:存储器的内容只能随机读出而不能写入。这类存储器常用来存放那些不能改变的信息。由于信息一旦写入存储器就固化了,即使断电,其写入的内容也不会丢失,所以ROM又称为固定存储器。ROM除了存放某些系统程序(如BIOS程序)外,还用来存放专用的子程序,或用作函数发生器、字符发生器及微程序控制器中的控制存储器。

(3) 顺序存取存储器(Sequential Access Memory,SAM)

SAM的存取方式与前两种存储器完全不同。SAM的内容只能按某种顺序存取,存取时间的长短与信息在存储体上的物理位置有关,所以SAM只能用平均存取时间作为衡量存取速度的指标。磁带机就是这样一类存储器。

(4) 直接存取存储器(Direct Access Memory,DAM)

DAM既不像RAM那样能随机地访问任一个存储单元,也不像SAM那样完全按顺序存取,而是介于两者之间。当要存取所需的信息时,第一步直接指向整个存储器中的某个小区域(如磁盘上的磁道);第二步在小区域内顺序检索或等待,直至找到目的地后再进行读/写操作。这种存储器的存取时间也是与信息所在的物理位置有关的,但比SAM的存取时间要短。磁盘机就属于这类存储器。

由于SAM和DAM的存取时间都与存储体的物理位置有关,所以又可以把它们统称为串行访问存储器。

3. 按存储介质分类

(1) 磁芯存储器

采用具有矩形磁滞回线的铁氧体磁性材料,利用两种不同的剩磁状态表示"1"或"0"。一颗磁芯存放一个二进制位,成千上万颗磁芯组成磁芯体。磁芯存储器的特点是信息可以长期存储,不会因断电而丢失;但磁芯存储器的读出是破坏性读出,即不论磁芯原存的内容为"0"还是"1",读出之后磁芯的内容一律变为"0",因此需要再重写一次,这就额外地增加了操作时间。从20世纪50年代开始,磁芯存储器曾一度成为主存的主要存储介质,但因磁芯存储器容量小、速度慢、体积大、可靠性低,从20世纪70年代开始,已经被半导体存储器逐渐取代。

(2) 半导体存储器

采用半导体器件制造的存储器,主要有MOS型存储器和双极型(TTL电路或ECL电路)存储器两类。MOS型存储器具有集成度高、功耗低、价格便宜、存取速度较慢等特点;双极型存储器具有存取速度快、集成度较低、功耗较大、成本较高等特点。半导体RAM存储的信息会因为断电而丢失。

(3) 磁表面存储器

在金属或塑料基体上,涂覆一层磁性材料,用磁层存储信息,常见的有磁盘、磁带等。由于它的容量大、价格低、存取速度慢,故多用作辅助存储器。

(4) 光存储器

采用激光技术控制访问的存储器,一般分为只读式、一次写入式、可改写式3种,它们的存储容量都很大,是目前使用非常广泛的辅助存储器。

4. 按信息的可保存性分类

断电后存储信息即消失的存储器称为易失性存储器,例如半导体RAM。断电后信息仍然保存的存储器称为非易失性存储器,例如ROM、磁芯存储器、磁表面存储器和光存储器。

如果某个存储单元所存储的信息被读出时,原存信息将被破坏,则称破坏性读出;如果读出时,被读单元原存信息不被破坏,则称非破坏性读出。具有破坏性读出性能的存储器,每当一次读出操作之后,必须紧接一个重写(再生)的操作,以便恢复被破坏的信息。

从原理上讲,只要具有两种明显稳定的物理状态的器件和介质都能用来存储二进制信息,但真正能用来做存储器的器件和介质还需要满足各类存储器技术指标的要求。

5.1.2 存储系统层次结构

为了解决存储容量、存取速度和价格之间的矛盾,通常把各种不同存储容量、不同存取速度的存储器按一定的体系结构组织起来,形成一个统一整体的存储系统。

多级存储层次如图5-1所示。从CPU的角度来看,n种不同的存储器($M_1 \sim M_n$)在逻辑上是一个整体。其中,M_1速度最快、容量最小、位价格最高;M_n速度最慢、容量最大、位价格最低。整个存储系统具有接近于M_1的速度,相等或接近M_n的容量,接近于M_n的位价格。在多级存储层次中,最常用的数据在M_1中,次常用的在M_2中,最少使用的在M_n中。

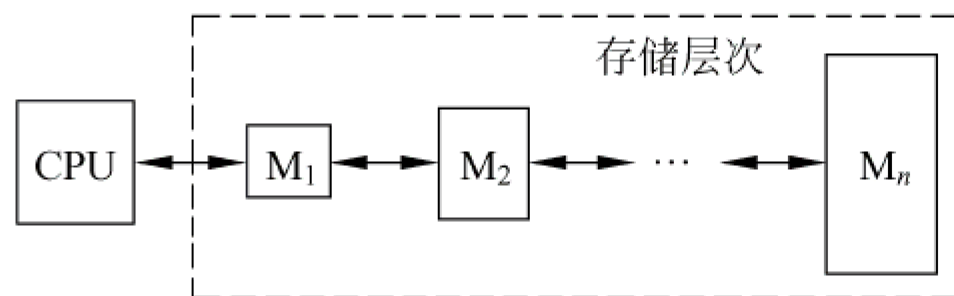


图5-1 多级存储层次

对于由M_1和M_2构成的两级存储层次结构,假设M_1、M_2访问时间分别为T_{A1}、T_{A2}。

命中率H定义为CPU产生的逻辑地址能在M_1中访问到的概率。在一个程序执行期间,设N_1为访问M_1的次数,N_2为访问M_2的次数,则命中率H为:

$$H = \frac{N_1}{N_1 + N_2}$$

不命中率或失效率是指由CPU产生的逻辑地址在M_1中访问不到的概率。对于两级存储层次,失效率为$1-H$。

两级存储层次的等效访问时间T_A根据M_2的启动时间有:

假设M_1访问和M_2访问是同时启动的,$T_A = H \times T_{A1} + (1-H) \times T_{A2}$。

假设M_1不命中时才启动M_2,$T_A = H \times T_{A1} + (1-H) \times (T_{A1} + T_{A2}) = T_{A1} + (1-H) \times T_{A2}$。

存储层次的访问效率：

$$e = \frac{T_{A1}}{T_A}$$

由高速缓冲存储器、主存储器、辅助存储器构成的三级存储系统可以分为两个层次，其中高速缓存和主存间称为Cache-主存存储层次(Cache存储系统)，如图5-2(a)所示；主存和辅存间称为主存-辅存存储层次(虚拟存储系统)，如图5-2(b)所示。

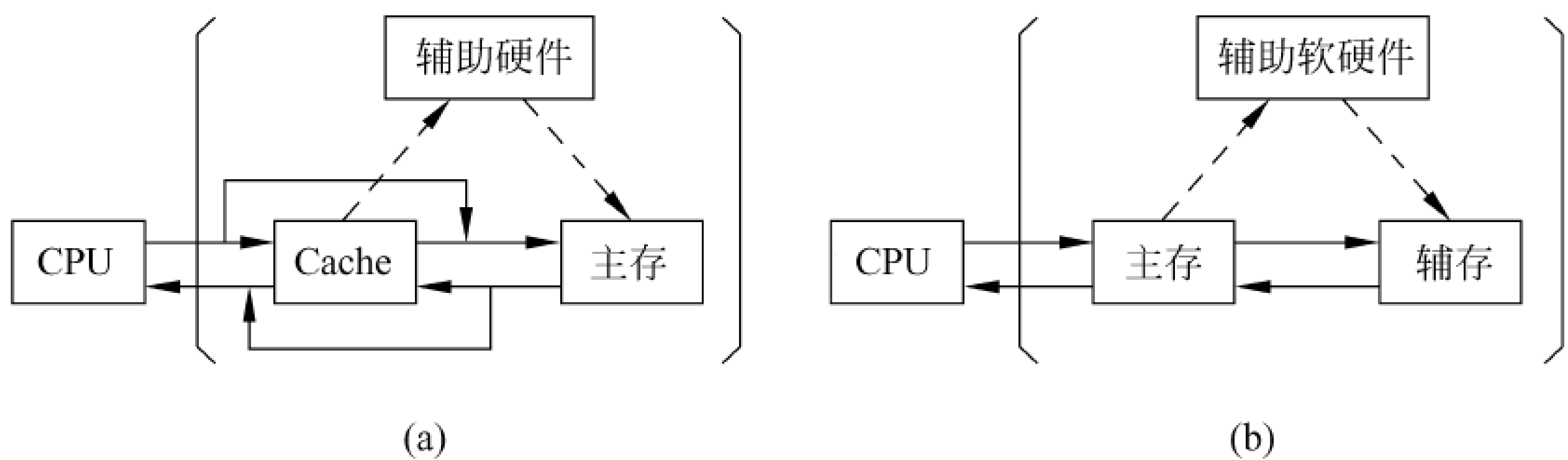


图5-2　两种存储层次

Cache存储系统是为解决主存速度不足而提出来的。在Cache和主存之间，增加辅助硬件，让它们构成一个整体。从CPU看，速度接近Cache的速度，容量是主存的容量，每位价格接近于主存的价格。由于Cache存储系统全部用硬件来调度，因此它对系统程序员和应用程序员都是透明的。

虚拟存储系统是为解决主存容量不足而提出来的。在主存和辅存之间，增加辅助的软硬件，让它们构成一个整体。从CPU看，速度接近主存的速度，容量是虚拟的地址空间，每位价格接近于辅存的价格。由于虚拟存储系统需要通过操作系统来调度，因此对系统程序员是不透明的，但对应用程序员是透明的。

5.2　主存储器的组织

主存储器是整个存储系统的核心，它用来存放计算机运行期间所需要的程序和数据，CPU可直接随机地对它进行访问。

5.2.1　主存储器的基本结构

主存通常由存储体、地址译码(address decoding)驱动电路、I/O和读写电路组成，其组成框图如图5-3所示。

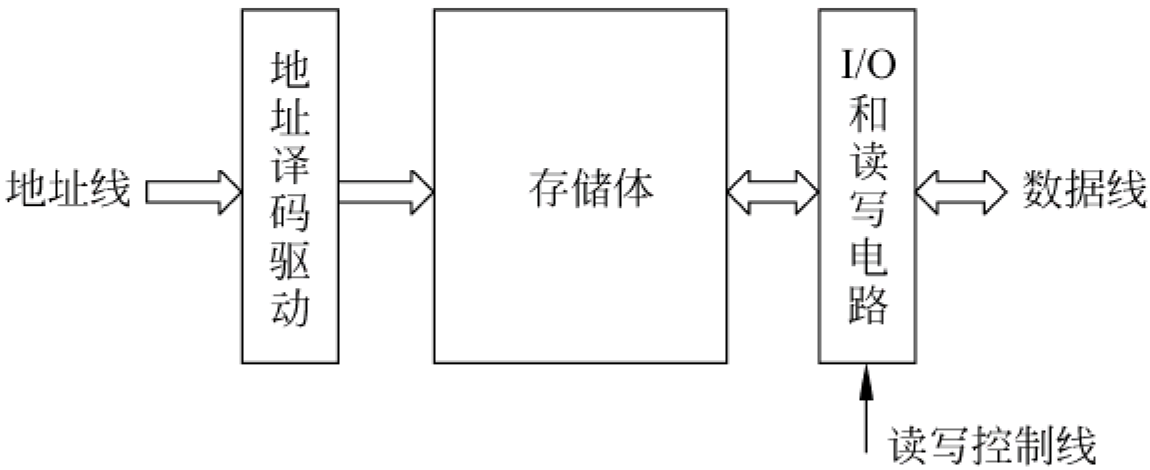


图5-3　主存的组成框图

存储体是主存储器的核心，程序和数据都存放在存储体中。

地址译码驱动电路实际上包含译码器和驱动器两部分。译码器将地址总线输入的地址码转换成与之对应的译码输出线上的有效电平，以表示选中了某一存储单元，然后由驱动器提供驱动电流去驱动相应的读写电路，完成对被选中存储单元的读写操作。

I/O 和读写电路包括读出放大器、写入电路和读写控制电路，用以完成被选中存储单元中各位的读出和写入操作。

主存的读写操作是在控制器的控制下进行的，只有接收到来自控制器的读写命令或写允许信号后，才能实现正确的读写操作。

5.2.2 主存储器的存储单元

位是二进制数的最基本单位，也是存储器存储信息的最小单位。一个二进制数由若干位组成，当这个二进制数作为一个整体存入或取出时，这个数称为存储字。存放存储字或存储字节的主存空间称为存储单元或主存单元，大量存储单元的集合构成一个存储体，为了区别存储体中的各个存储单元，必须将它们逐一编号。存储单元的编号称为地址，地址和存储单元之间有一对一的对应关系，就像一座大楼的每个房间都有房间号一样。

一个存储单元可能存放一个字，也可能存放一个字节，这是由计算机的结构确定的。对于字节编址的计算机，最小寻址单位是一个字节，相邻的存储单元地址指向相邻的存储字节；对于字编址的计算机，最小寻址单位是一个字，相邻的存储单元地址指向相邻的存储字。所以，存储单元是 CPU 对主存可访问操作的最小存储单位。

例如，IBM 370 机是字长为 32 位的计算机，主存按字节编址，每一个存储字包含 4 个单独编址的存储字节，其地址安排如图 5-4(a)所示。它被称为大端(big-endian)方案，即字地址等于最高有效字节地址，且字地址总是等于 4 的整数倍，正好用地址码的最末两位来区分同一个字的 4 个字节。PDP-11 机是字长为 16 位的计算机，主存也按字节编址，每一个存储字包含 2 个单独编址的存储字节，其地址安排如图 5-4(b)所示。它被称为小端(little-endian)方案，即字地址等于最低有效字节地址，且字地址总是等于 2 的整数倍，正好用地址码的最末一位来区分同一个字的 2 个字节。从图 5-4 可以看出，大端方案从最高有效字节向最低有效字节进行字节地址编号，小端方案从最低有效字节向最高有效字节进行字节地址编号。

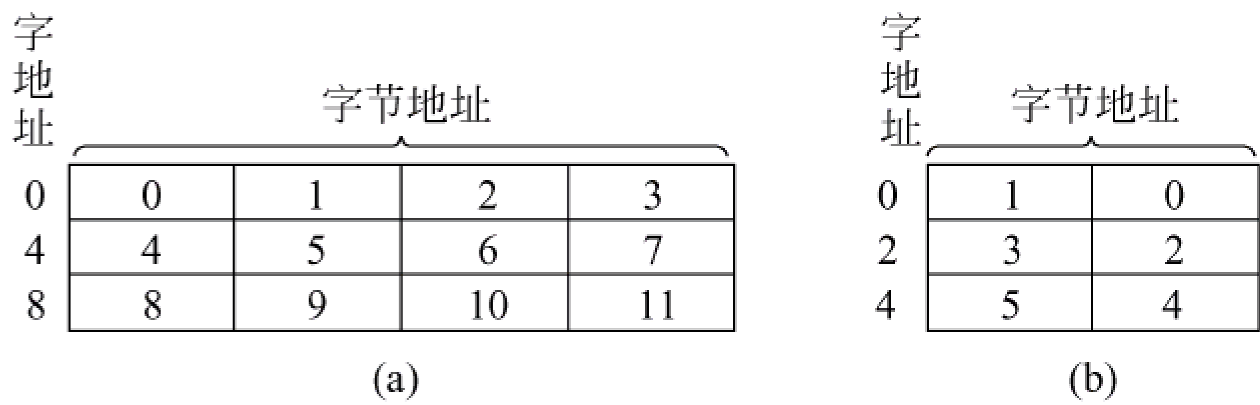


图 5-4 字节编址计算机的地址安排方案

采用大、小端方案对数据进行存放的主要区别在于存放的字节顺序，如果将一个 32 位的整数 0x12345678 存放到一个整型变量(int)中，这个整型变量采用大端或者小端方案在主存中的存放如表 5-1 所示(假设从地址 0x4000 开始存放)。

表 5-1　32 位整型变量存放方案

主存地址	大端方案	小端方案
0x4000	12(MSB)	78(LSB)
0x4001	34	56
0x4002	56	34
0x4003	78(LSB)	12(MSB)

从表 5-1 可以看出,大端方案将高字节(MSB)存放在低地址,小端方案将高字节存放在高地址。采用大端方案进行数据存放符合人类的正常思维,而采用小端方案进行数据存放利于计算机处理。到目前为止,采用大端或者小端进行数据存放,其孰优孰劣也没有定论。大端方案与小端方案的差别体现在一个处理器的寄存器、指令集、数据总线等各个层次中。

Intel 80x86 采用小端方案,IBM 370、Motorola 680x0 和大多数 RISC 机器则采用大端方案。有些微处理器,如 Power PC,既支持大端方案又支持小端方案,在芯片加电启动时只要选择确定即可(默认为大端方案,可选择小端方案)。

在排序顺序不同的系统之间进行数据通信时,需要进行顺序转换。了解字节顺序的好处在于调试底层机器级程序时,能够清楚每个数据的字节顺序,以便将一个机器数正确转化为真值。

5.2.3　主存储器的主要技术指标

1. 存储容量

对于字节编址的计算机,以字节数来表示存储容量;对于字编址的计算机,以字数与其字长的乘积来表示存储容量。例如,某计算机的主存容量为 64K×16,表示它有 64K 个存储单元,每个存储单元的字长为 16 位,若改用字节数表示,则可记为 128K 字节(即 128KB)。

2. 存取速度

主存的存取速度通常由存取时间 T_a、存取周期 T_m 和主存带宽 B_m 等参数来描述。

(1) 存取时间 T_a

存取时间又称为访问时间或读写时间,它是指从启动一次存储器操作到完成该操作所经历的时间。例如,读出时间是指从 CPU 向主存发出有效地址和读命令开始,直到将被选单元的内容读出为止所用的时间;写入时间是指从 CPU 向主存发出有效地址和写命令开始,直到信息写入被选中单元为止所用的时间。显然,T_a 越小,存取速度越快。

(2) 存取周期 T_m

存取周期又称读写周期、访存周期,是指主存进行一次完整的读写操作所需的全部时间,即连续两次访问存储器操作之间所需要的最短时间。显然,一般情况下,$T_m > T_a$。这是因为对于任何一种存储器,在读写操作之后,总要有一段恢复内部状态的复原时间。对于破坏性读出的 RAM,存取周期往往比存取时间要大得多,甚至可以达到 $T_m = 2T_a$,这是因为存储器中的信息读出后需要马上进行重写(再生)。

(3) 主存带宽 B_m

与存取周期密切相关的指标是主存的带宽,它又称为数据传输率,表示每秒从主存进出信息的最大数量,单位为字每秒或字节每秒或位每秒。主存带宽与主存的等效工作频率及主存位宽有关系,若单位为字节每秒,则有:

$$B_m = \text{主存等效工作频率} \times \text{主存位宽} \div 8$$

目前,主存提供信息的速度还跟不上 CPU 处理指令和数据的速度,所以,主存的带宽是改善计算机系统瓶颈的一个关键因素。为了提高主存的带宽,可以采取的措施有:

- 缩短存取周期。
- 增加存储字长。
- 增加存储体。

3. 可靠性

可靠性是指在规定的时间内,存储器无故障读写的概率。通常,用平均故障间隔时间(Mean Time Between Failures,MTBF)来衡量可靠性。MTBF 越长,说明存储器的可靠性越高。

4. 功耗

功耗是一个不可忽视的问题,它反映了存储器件耗电的多少,同时也反映了其发热的程度。通常希望功耗要小,这对存储器件的工作稳定性有好处。大多数半导体存储器的工作功耗与维持功耗是不同的,后者大大地小于前者。

5.2.4 数据在主存中的存放

目前,大多数存储器采用字节编址,数据在主存中有 3 种不同存放方法,如图 5-5 所示。设存储字长为 64 位(8 个字节),即一个存取周期最多能够从主存读或写 64 位数据。图 5-5 中最左边一列表示字地址(十六进制),字地址的最末 3 个二进制位必定为 000。假设读写的数据有 4 种不同长度,它们分别是字节(8 位)、半字(16 位)、单字(32 位)和双字(64 位)。

注意:此例中数据字长(32 位)不等于存储字长(64 位)。

图 5-5(a)是一种不浪费存储器资源的存放方法,4 种不同长度的数据一个紧接着上一个存放。这种数据存放方式的优点是不浪费宝贵的主存资源。但是存在问题,主要问题有两个:一是除了访问一个字节以外,当要访问一个双字、一个单字或一个半字时都有可能需要花费两个存取周期,因为从图 5-5(a)中可以看出,一个双字、一个单字或一个半字都有可能跨越两个存储字存放,这使存储器的工作速度降低了一半;二是存储器的读写控制比较复杂。

为了克服上述两个缺点,出现了图 5-5(b)所示的另一种数据存放方法。这种存放方法规定,无论要存放的是字节、半字、单字或双字,都必须从一个存储字的起始位置开始存放(在图 5-5(b)中是从最左边放起),而多余的部分浪费不用。这种数据存放方法的优点是:无论访问一个字节、一个半字、一个单字或一个双字都可以在一个存储周期内完成,读写数据的控制比较简单。但它的主要缺点是浪费了宝贵的存储器资源,如果双字、单字、半字、字节 4 种不同长度的数据出现的概率相同的话,那么主存的实际利用率只有约 50%,即有一

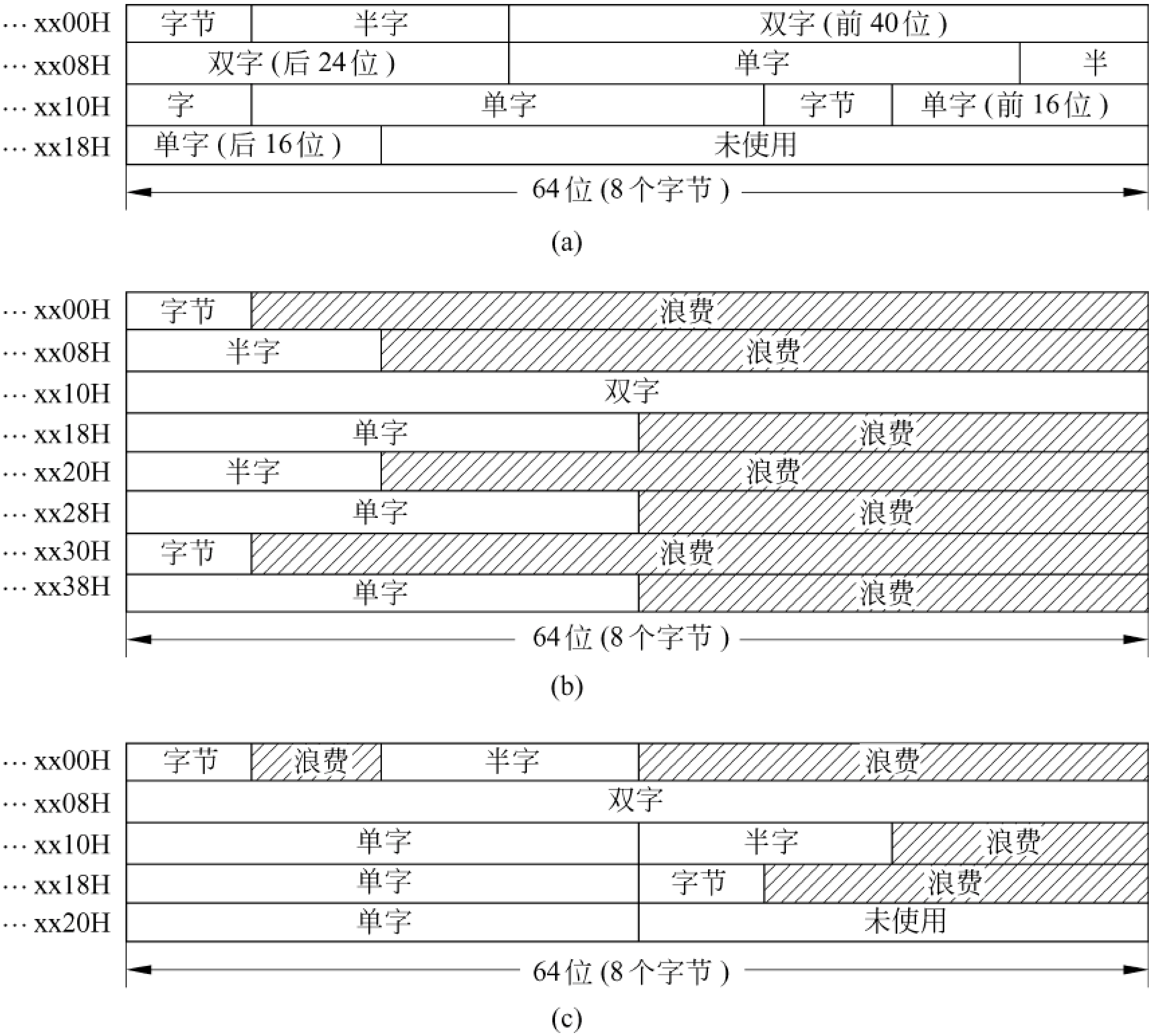


图5-5　字节编址的主存储器的各种存放方法

半的存储空间被浪费。

综合前两种数据存放方法的优缺点,出现了如图5-5(c)所示的折中方案。图5-5(c)所示的存放方法规定,双字数据(8个字节)的起始地址的最末3个二进制位必须为000(8的整倍数),单字数据(4个字节)的起始地址的最末两位必须为00(4的整倍数),半字数据(2个字节)的起始地址的最末一位必须为0(偶数)。这种存储方式能够保证无论访问双字、单字、半字或字节,都能在一个存取周期内完成,尽管存储器资源仍然有浪费,但是比图5-5(b)所示的存放方法要好得多。这种存放方法被称为边界对齐的数据存放方法。

5.3　半导体随机存储器和只读存储器

主存储器通常分为RAM和ROM两大部分。RAM可读可写,ROM只能读不能写。下面重点介绍RAM的工作原理与结构,以及ROM的基本类型。

5.3.1　RAM记忆单元电路

通常把存放一个二进制位的物理器件称为记忆单元,它是存储器的最基本构件,地址码相同的多个记忆单元构成一个存储单元。记忆单元可以由各种材料制成,但最常见的由MOS电路组成。RAM又可分为静态RAM(Static RAM,SRAM)和动态RAM(Dynamic RAM,DRAM)两种。

1. 6 管 SRAM 记忆单元电路

6 管 SRAM 记忆单元电路如图 5-6 所示。

SRAM 记忆单元是用双稳态触发器来记忆信息的，从图 5-6 中可以看出，T_1～T_6 管构成一个记忆单元的主体，能存放一位二进制信息。其中，T_1 和 T_2 管构成存储信息的双稳态触发器；T_3 和 T_4 管构成门控电路，控制读写操作；T_5 和 T_6 是 T_1 和 T_2 管的负载管。电路中有一条字线，用来选择这个记忆单元，还有两条位线，用来传送读写信号。

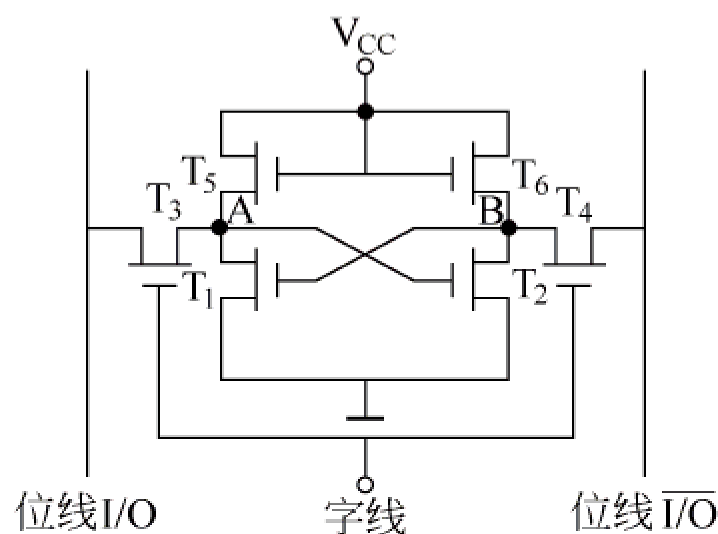


图 5-6　6 管 SRAM 记忆单元电路

假定：T_1 管截止、T_2 管导通（A＝1、B＝0）表示该记忆单元中存储的是“1”信息；T_1 管导通、T_2 管截止（A＝0、B＝1）表示该记忆单元中存储的是“0”信息。

当字线为低电平时，该记忆单元未被选中，T_3 和 T_4 管截止，触发器与位线隔开，原存信息不会改变，称为保持状态。

当字线为高电平时，该记忆单元被选中，T_3 和 T_4 管导通，可进行读写操作。位线 I/O 被称为读写“1”线，位线$\overline{I/O}$被称为读写“0”线。

读操作：因为 T_3 和 T_4 管导通，相当于 A 点和 B 点分别与位线 I/O 和$\overline{I/O}$相连。若记忆单元原存“1”，则 I/O 线输出高电平，$\overline{I/O}$线输出低电平，完成读“1”操作；若记忆单元原存“0”，则 I/O 线输出低电平，$\overline{I/O}$线输出高电平，完成读“0”操作。

写操作：如果要写入“1”，则在 I/O 线上输入高电平，$\overline{I/O}$线上输入低电平，它们将分别通过 T_3 和 T_4 管迫使 T_1 管截止、T_2 管导通，该记忆单元内容成为“1”，完成写“1”操作；如果要写入“0”，则在 I/O 线上输入低电平，$\overline{I/O}$线输入高电平，经过同样的路径迫使 T_1 管导通、T_2 管截止，该记忆单元内容成为“0”，完成写“0”操作。

在该记忆单元未被选中或读出时，电路处于双稳态触发器工作状态，由电源 V_{CC} 不断给 T_1 和 T_2 管供电，以保存信息。但是，只要电源被切断，原来的保存信息便会丢失，这就是半导体存储器的易失性。

SRAM 的存取速度快，但集成度低，功耗也较大，所以一般用来组成高速缓冲存储器和小容量主存系统。

2. 4 管 DRAM 记忆单元电路

如果将前述 6 管 SRAM 记忆单元电路中的两个负载管（T_5 和 T_6）去掉，便形成 4 管 DRAM 记忆单元电路，如图 5-7 所示。负载回路断开后，保持状态时没有外加电源供电，因而 T_1 和 T_2 管不再构成双稳态触发器，所以动态 MOS 记忆单元是靠 MOS 电路中的栅极电容 C_1 和 C_2 来存储信息的。

图 5-7 中虚线框外的两个 MOS 管是公用的预充管，不算在记忆单元电路中。

假定：C_2 上有电荷（高电平）、C_1 上无电荷（低电平）时，表示存储“1”信息；C_2 上无电荷（低电平）、C_1 上有电荷（高电平）时，表示存储“0”信息。由于 MOS 管栅极的输入电阻很大，所以栅极泄漏电流很小，即使没有负载管的供电，栅极电容上的电荷也能保存相当一段时

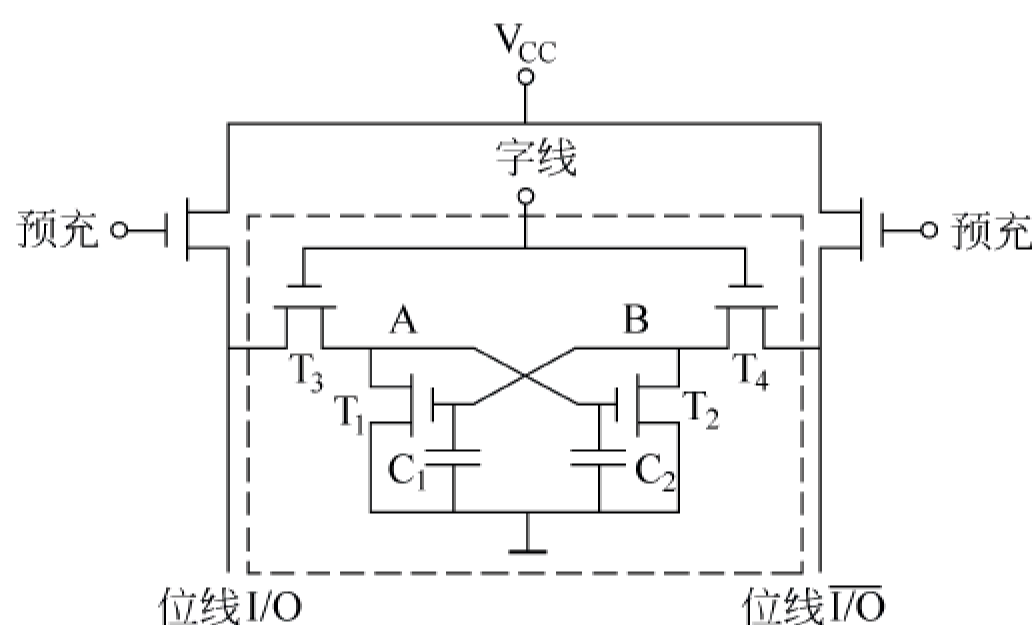


图 5-7　4 管 DRAM 记忆单元电路

间。然而,栅极电容上的电荷数目毕竟是有限的,虽然释放速度很慢,但经过一段时间后仍然会使记忆单元中存储的信息丢失,因此每隔一定的时间必须向栅极电容补充一次电荷,这个过程称为"刷新"。

平时记忆单元处于保持状态,字线为低电平,原存的信息被存储在 T_1 和 T_2 管的栅极分布电容 C_1 或 C_2 上,栅极的泄漏电流使存储的信息能保留几毫秒。

写入时,字线为高电平,打开选通门 T_3 和 T_4。如果写"1",在 I/O 线上输入高电平,$\overline{I/O}$线上输入低电平,使 C_2 充电、C_1 放电;如果写"0",在 I/O 线上输入低电平,$\overline{I/O}$线上输入高电平,使 C_1 充电、C_2 放电。

读出时,预充电信号先打开预充管,使位线 I/O 和$\overline{I/O}$均成为高电平。当选中记忆单元后,选通门被打开,若原存"1",则 C_2 上有电荷,T_2 导通,使$\overline{I/O}$线上有负脉冲输出,读出"1";若原存"0",则 C_1 上有电荷,T_1 导通,使 I/O 线上有负脉冲输出,读出"0"。在读出过程中,电源还通过 I/O 或$\overline{I/O}$线分别向 C_2 或 C_1 补充电荷。

4 管 DRAM 记忆单元的刷新过程也是对栅极电容 C_1 或 C_2 补充电荷的过程,因此刷新过程也就是读出过程,只是这种读出的目的不是为了从 I/O 线或$\overline{I/O}$线上得到读出信息,而是为了对记忆单元进行刷新操作,常将其称作"假读"。

DRAM 集成度高,功耗小,但存取速度慢,一般用来组成大容量主存系统。

3. 单管 DRAM 记忆单元电路

进一步减少记忆单元中 MOS 管的数目可形成更简单的 3 管 DRAM 记忆单元或单管 DRAM 记忆单元。这里仅介绍单管 DRAM 记忆单元,其电路结构如图 5-8 所示。

从图 5-8 中可看出,单管动态记忆单元由一个 MOS 管 T_1 和一个存储电容 C 构成。

当字线为高电平时,该电路被选中。

写入时,若写入"1",位线为高电平,对电容 C 充电;若写入"0",位线为低电平,C 上的电荷经位线泄放。

读出时,若原存"1",C 上有电荷,经 T_1 管在位线上产生读电流,完成读"1"操作;若原存"0",C 上无电荷,在位线上不产生读电流,完成读"0"操作。当读操作完毕,存储电容 C 上的电荷已被泄放完,故是破坏性读出,必须采取重写(再生)的措施。

存储电容 C 的容量不可能做得很大,一般比位线上的寄生电

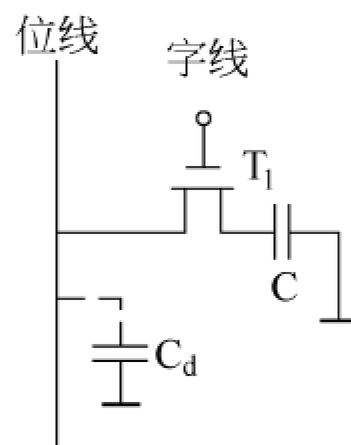


图 5-8　单管 DRAM 记忆单元电路

容 C_d 还要小。在读出时，T_1 导通后，电荷将在 C 和 C_d 间分配，就会使读出信息减小，所以，用单管记忆单元组成的存储器中，读出放大器应有较高的灵敏度。因为信息是存储在一个很小电容 C 上，也只能保留几毫秒的时间，所以必须定时地进行刷新操作。

显然，单管 DRAM 记忆单元与 4 管 DRAM 记忆单元比较，具有功耗更小、集成度更高的优点。

5.3.2 动态 RAM 的刷新

1. 刷新间隔

前面已经说过，为了维持 DRAM 记忆单元的存储信息，每隔一定时间必须刷新。那么每隔多少时间进行一次刷新操作呢？这主要是根据栅极电容上电荷的泄放速度来决定的。一般选定的最大刷新间隔为 2ms 或 4ms 甚至更大，也就是说，应在规定的时间内，将全部存储体刷新一遍。

注意：*刷新和重写（再生）是两个完全不同的概念，切不可混淆。重写是随机的，某个存储单元只有在破坏性读出之后才需要重写。而刷新是定时的，即使许多记忆单元长期未被访问，若不及时补充电荷的话，信息也会丢失。重写一般是按存储单元进行的，而刷新通常是以存储体矩阵中的一行为单位进行的。*

2. 刷新方式

常见的刷新方式有集中式、分散式和异步式 3 种。

(1) 集中刷新方式

在允许的最大刷新间隔（如 2ms）内，按照存储芯片容量的大小集中安排若干个刷新周期，刷新时停止读写操作。

刷新时间＝存储矩阵行数×刷新周期

这里刷新周期是指刷新一行所需要的时间，由于刷新过程就是“假读”的过程，所以刷新周期就等于存取周期。

例如，对具有 1024 个记忆单元（排列成 32×32 的存储矩阵）的存储芯片进行刷新，刷新是按行进行的，且每刷新一行占用一个存取周期，所以共需 32 个周期以完成全部记忆单元的刷新。假设存取周期为 500ns(0.5μs)，则在 2ms 内共可以安排 4000 个存取周期，从 0～3967 个周期内进行读写操作或保持，而从 3968～3999 最后 32 个周期集中安排刷新操作，如图 5-9 所示。

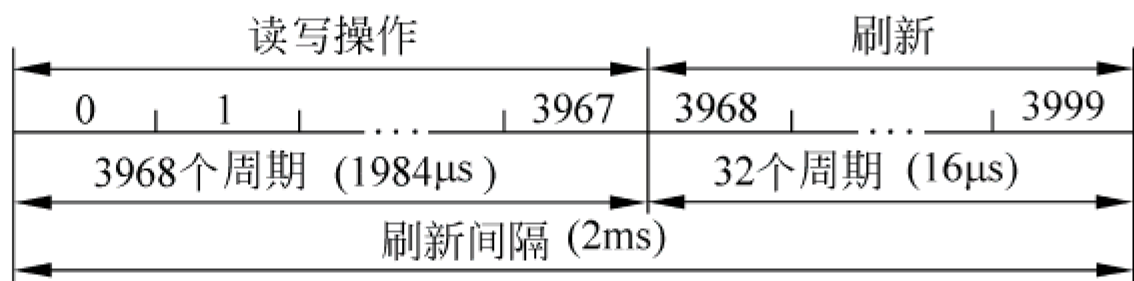


图 5-9　集中刷新方式示意图

集中刷新方式的优点是读写操作时不受刷新工作的影响，因此系统的存取速度比较高。主要缺点是在集中刷新期间必须停止读写，这一段时间称为“死区”，而且存储容量越大，死

区就越长。

(2) 分散刷新方式

分散刷新是指把刷新操作分散到每个存取周期内进行,此时系统的存取周期被分为两部分,前一部分时间进行读写操作或保持,后一部分时间进行刷新操作。在一个系统存取周期内刷新存储矩阵中的一行。

这种刷新方式增加了系统的存取周期,若存储芯片的存取周期为0.5μs,则系统的存取周期应为1μs,即前一个0.5μs读写,后一个0.5μs刷新。仍以前述的32×32矩阵为例,整个存储芯片刷新一遍需要32μs,如图5-10所示。

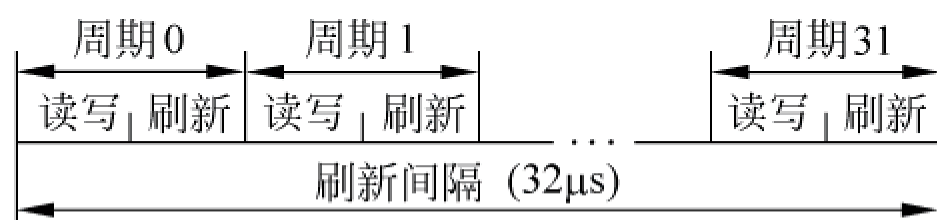


图5-10 分散刷新方式示意图

从图5-10中可以看出,这种刷新方式没有死区,但是,它也有很明显的缺点:一是加长了系统的存取周期,降低了整机的速度;二是刷新过于频繁(本例中每32μs就重复刷新一遍),尤其是当存储容量比较小的情况下,没有充分利用所允许的最大刷新间隔(2ms)。

(3) 异步刷新方式

异步刷新方式可以看成前述两种方式的结合,它充分利用了最大刷新间隔时间,把刷新操作平均分配到整个最大刷新间隔时间内进行,故有:

相邻两行的刷新间隔=最大刷新间隔时间÷行数

对于32×32矩阵,在2ms内需要将32行刷新一遍,所以相邻两行的刷新时间间隔=2ms÷32=62.5μs,即每隔62.5μs安排一个刷新周期。在刷新时封锁读写,如图5-11所示。

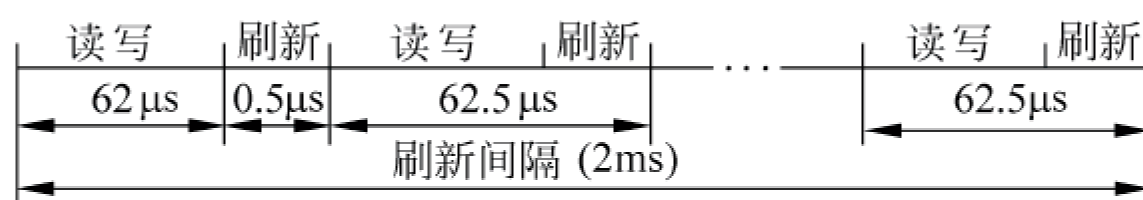


图5-11 异步刷新方式示意图

异步刷新方式虽然也有死区,但比集中刷新方式的死区小得多,仅为0.5μs。这样可以避免使CPU连续等待过长的时间,而且减少了刷新次数,是比较实用的一种刷新方式。

消除"死区"的方法,还可采用不定期的刷新方式。其基本做法是:把刷新操作安排在CPU不访问存储器的空闲时间里,如利用CPU取出指令后进行译码的这段时间。这种方式既不会出现死区,又不会降低存储器的存取速度;但是控制比较复杂,实现起来比较困难。

3. 刷新控制

为了控制刷新,往往需要增加刷新控制电路。刷新控制电路的主要任务是解决刷新和CPU访问存储器之间的矛盾。通常,当刷新请求和访存请求同时发生时,应优先进行刷新操作。也有些DRAM芯片本身具有自动刷新功能,即刷新控制电路在芯片内部。

DRAM的刷新要注意以下问题:

① 无论是由外部刷新控制电路产生刷新地址逐行循环地刷新,还是芯片内部的刷新地

址计数器自动地控制刷新，都不依赖于外部的访问，刷新对 CPU 是透明的。

② 刷新通常是一行一行地进行的，每一行中各记忆单元同时被刷新，故刷新操作时仅需要行地址，不需要列地址。

③ 刷新操作类似于读出操作，但又有所不同。因为刷新操作仅是给栅极电容补充电荷，不需要信息输出。另外，刷新时不需要加片选信号，即整个存储器中的所有芯片同时被刷新。

④ 因为所有芯片同时被刷新，所以在考虑刷新问题时，应当从单个芯片的存储容量着手，而不是从整个存储器的容量着手。

5.3.3 RAM 芯片分析

1. RAM 芯片

RAM 芯片通过地址线、数据线和控制线与外部连接。地址线是单向输入的，其数目与芯片容量有关。例如，容量为 1024×4[①] 时，地址线有 10 根；容量为 64K×1 时，地址线有 16 根。数据线是双向的，既可输入，也可输出，其数目与数据位数有关。例如，1024×4 的芯片，数据线有 4 根；64K×1 的芯片，数据线只有 1 根。控制线主要有读写控制线和片选线两种，读写控制线用来控制芯片是进行读操作还是写操作的，片选线用来决定该芯片是否被选中。各种 RAM 芯片的外引脚主要有：

- 地址线——A_i。
- 数据线——D_i。
- 片选线——$\overline{CE}$(或$\overline{CS}$)。
- 读写控制线——$\overline{WE}$或$\overline{OE}/\overline{WE}$。
- V_{CC}——+5V，工作电源。
- GND——地。

有些 SRAM 芯片有两根读写控制线：读允许线$\overline{OE}$和写允许线$\overline{WE}$。有些 SRAM 芯片只有 1 根读写控制线：$\overline{WE}$，当$\overline{WE}=0$ 时，写允许；$\overline{WE}=1$ 时，读允许。

由于 DRAM 芯片集成度高，容量大，为了减少芯片引脚数量，DRAM 芯片把地址线分成相等的两部分，分两次从相同的引脚送入。两次输入的地址分别称为行地址和列地址，行地址由行地址选通信号(Row Address Select，$\overline{RAS}$)送入存储芯片，列地址由列地址选通信号(Column Address Select，$\overline{CAS}$)送入存储芯片。由于采用了地址复用技术，因此，DRAM 芯片每增加一条地址线，实际上是增加了两位地址，也即增加了 4 倍的容量。

在 DRAM 芯片中，可以不设专门的片选线$\overline{CE}$，而用行选通信号$\overline{RAS}$、列选通$\overline{CAS}$兼作片选信号。

2. 地址译码方式

RAM 芯片中的地址译码电路能把地址线送来的地址信号翻译成对应存储单元的选择信号。地址译码方式有单译码和双译码两种。

① 通常用 $M\times N$ 来描述存储器或存储芯片的规格，其中 M 表示存储单元数，N 表示每个单元位数。

(1) 单译码方式

单译码方式又称字选法,所对应的存储器是字结构的。容量为 M 个字的存储器(M 个字,每字 b 位),排列成 M(行)×b(列)的矩阵,矩阵的每一行对应一个字,有一条公用的选择线 w_i,称为字线。地址译码器集中在水平方向,K 位地址线可译码变成 2^K 条字线,$M=2^K$。字线选中某个字长为 b 位的存储单元,经过 b 根位线可读出或写入 b 位存储信息。在图 5-12 所示结构图中有 $2^5\times8=256$ 个记忆单元,排列成 32 个字,每个字长 8 位。图中有 5 条地址线,经过译码产生 32 条字线 $w_0\sim w_{31}$。某一字线被选中时,同一行中的各位 $b_0\sim b_7$ 就都被选中,由读写电路对各位实施读出或写入操作。

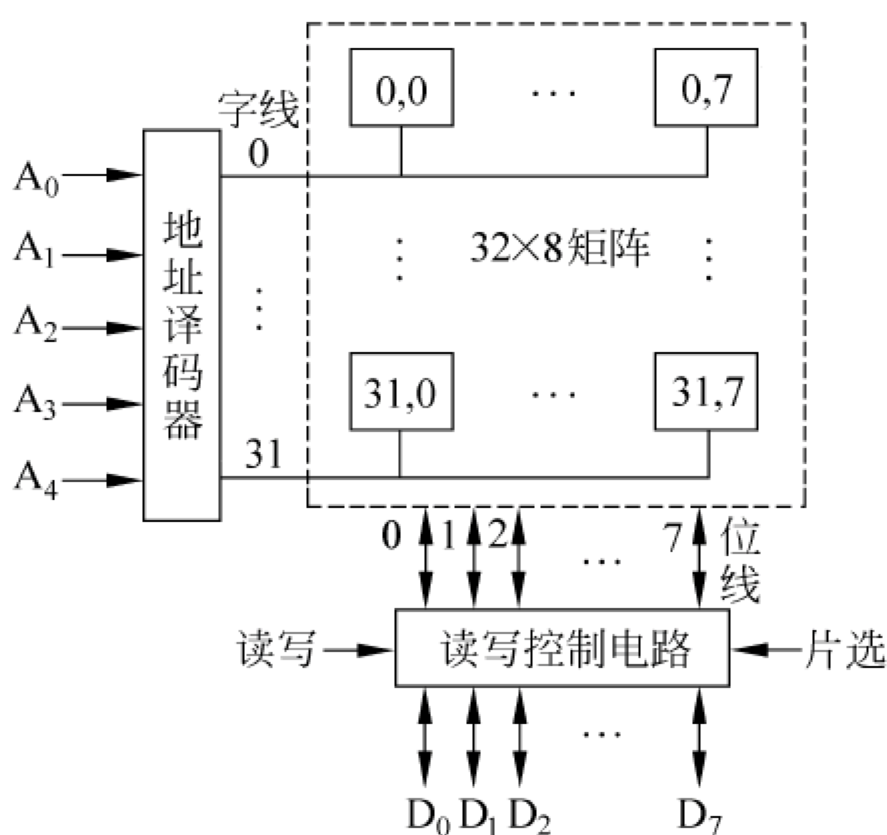


图 5-12 字结构、单译码方式 RAM

字结构的优点是结构简单,缺点是使用的外围电路多,成本昂贵。更严重的是,当字数大大超过位数时,存储体会形成纵向很长而横向很窄的不合理结构,所以这种方式只适用于容量不大的存储器。

(2) 双译码方式

双译码方式又称为重合法。通常是把 K 位地址线分成接近相等的两段,一段用于水平方向作 X 地址线,供 X 地址译码器译码;一段用于垂直方向作 Y 地址线,供 Y 地址译码器译码。X 和 Y 两个方向的选择线在存储体内部的每个记忆单元上交叉,以选择相应的记忆单元。

双译码方式对应的存储芯片结构可以是位结构的,也可以是字段结构的。对于位结构的存储芯片,容量为 $M\times1$,把 M 个记忆单元排列成存储矩阵(尽可能排列成方阵)。图 5-13 所示结构是 4096×1,排列成 64×64 的矩阵。地址码共 12 位,X 方向和 Y 方向各 6 位。若要组成一个 M 字×b 位的存储器,就需要把 b 片 $M\times1$ 的存储芯片并列连接起来,即在图 5-13 所示 Z 方向上重叠 b 个芯片。

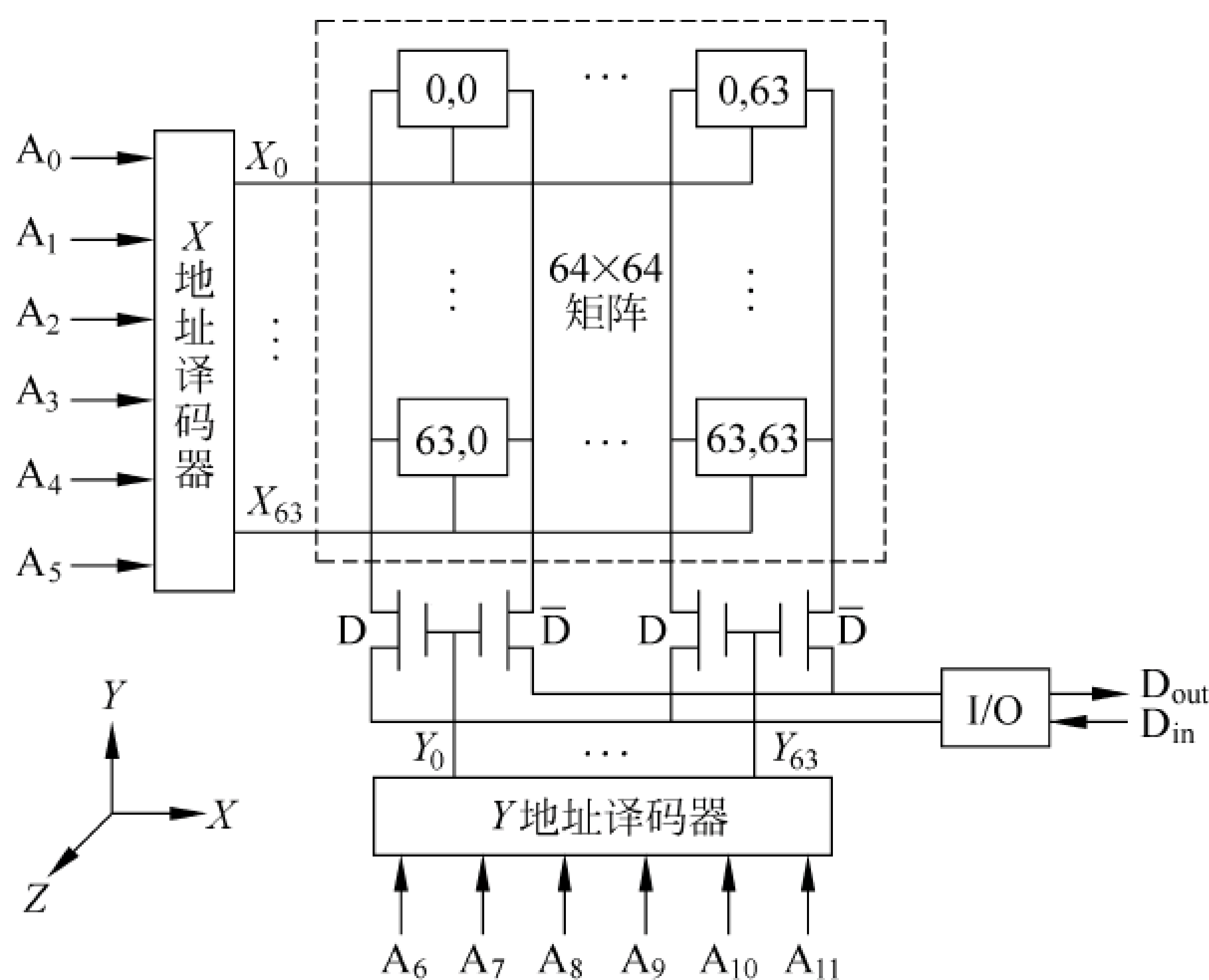


图 5-13 位结构、双译码方式 RAM

对于字段结构的存储芯片，在一根行选择线上安排的不是一个 b 位长的字，而是 s 个 b 位长的字。这将使行选择线减为 M/s 根，列选择线数为 s，而每一条列选线同时选择 b 位数据，从而使存储芯片的物理结构大大改进，接近或成为方阵。K 位地址线也要划分为两部分：$K_x=\log_2 M/s$，$K_y=\log_2 s$。1K×4 的芯片中共有 4096 个记忆单元，分成 64×64 的方阵，6 位地址线经 X 地址译码器形成 64 根行选择线，剩下的 4 位地址线经 Y 地址译码器形成 16 根列选择线，每条列选择线同时选择 4 位数据。

典型的 RAM 芯片中的记忆单元总数往往开方之后仍是一个常数，如 1K×1，1K×4，2K×8，4K×1，4K×4，8K×8，16K×1，64K×1 等，也就是使存储体成为一个方阵。

双译码方式与单译码方式相比，减少了选择线数目和驱动器数目。例如，存储容量 $N=2^{16}=64K$ 单元，两种译码方式的比较如表 5-2 所示。存储容量越大，这两种方式的差异越明显。

表 5-2　两种译码方式比较

译码方式	占用地址位		选择线数		驱动器数	
单译码	16		65 536		65 536	
双译码	8	8	256	256	256	256

3. RAM 的读写时序

（1）SRAM 读写时序

图 5-14(a)为典型的读周期时序，读周期表示对该芯片进行两次连续读操作的最小间隔时间。在此期间，地址输入信息不允许改变，片选信号$\overline{CS}$在地址有效之后变为有效，使芯片被选中，最后在数据线上得到读出的信号。写允许信号$\overline{WE}$在读周期中保持高电平。

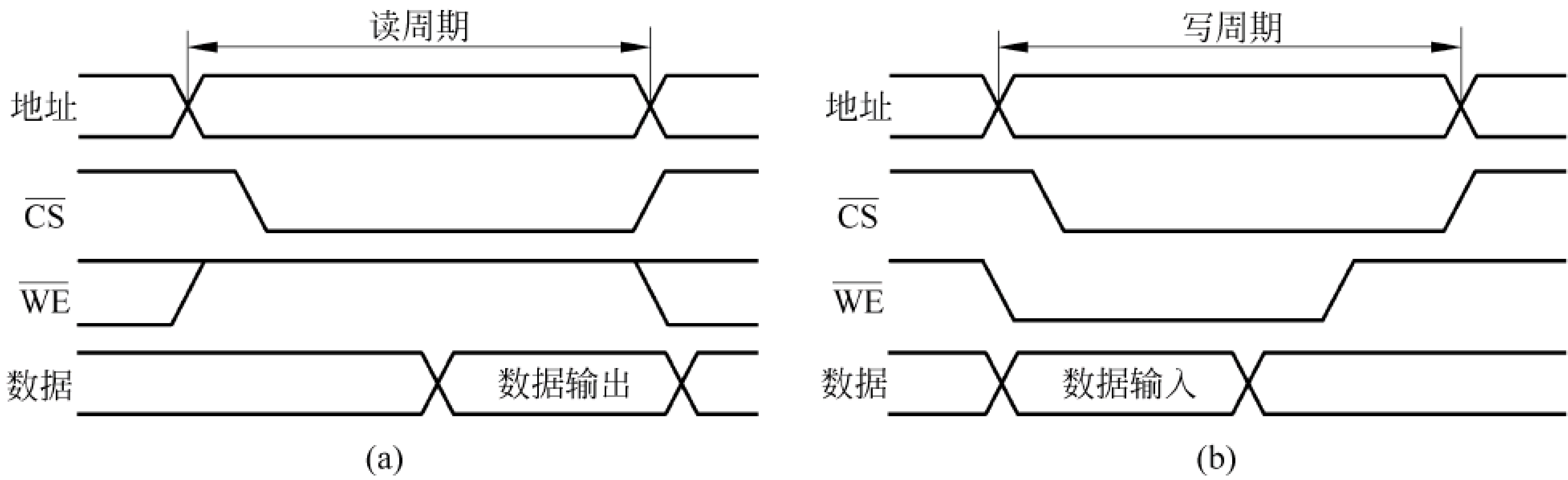


图 5-14　静态 RAM 的读写时序

图 5-14(b)为典型的写周期时序，它与读周期相似，但除了要有地址和片选信号外，还要加一个低电平有效的写入脉冲$\overline{WE}$，并提供写入数据。

（2）DRAM 读写时序

DRAM 的读周期和写周期时序图分别如图 5-15(a)和图 5-15(b)所示。

在一个读周期中，行地址必须在$\overline{RAS}$有效之前有效，列地址也必须在$\overline{CAS}$有效之前有效，且在$\overline{CAS}$到来之前，$\overline{WE}$必须为高电平，并保持到$\overline{CAS}$脉冲结束之后。

在一个写周期中，当$\overline{WE}$有效之后，输入的数据必须保持到$\overline{CAS}$变为低电平之后。在

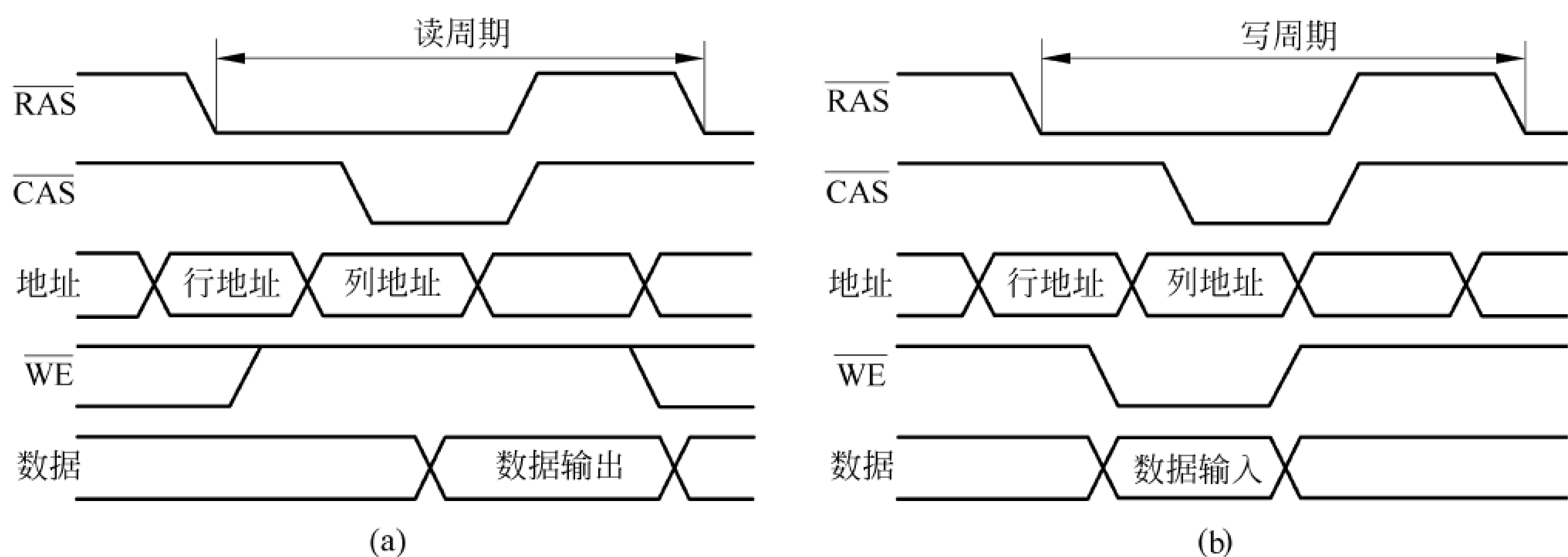


图 5-15 动态 RAM 的读写时序

$\overline{RAS}$、$\overline{CAS}$和$\overline{WE}$全部有效时,数据被写入存储器。

5.3.4 半导体只读存储器

ROM 的最大优点是具有非易失性,即使电源断电,ROM 中存储的信息也不会丢失。

1. ROM 的类型

ROM 工作时只能读出,不能写入,那么 ROM 中的内容是如何事先存入的呢?把向 ROM 写入数据的过程称为对 ROM 进行编程,根据编程方法的不同,ROM 通常可以分为以下几类。

(1) 掩膜式 ROM(MROM)

它的内容是由半导体制造厂按用户提出的要求在芯片的生产过程中直接写入的,写入之后任何人都无法改变其内容。

MROM 的优点是可靠性高,集成度高,形成批量之后价格便宜;缺点是用户对制造厂的依赖性过大,灵活性差。

(2) 一次可编程 ROM(PROM)

PROM 允许用户利用专门的设备(编程器)写入自己的程序,一旦写入,其内容将无法改变。

PROM 产品出厂时,所有记忆单元均制成“0”(或制成“1”),用户根据需要可自行将其中某些记忆单元改为“1”(或改为“0”)。双极型 PROM 有两种结构,一种是熔丝烧断型,另一种是 PN 结击穿型,由于它们的写入都是不可逆的,所以只能进行一次性写入。

(3) 可擦除可编程 ROM(EPROM)

EPROM 不仅可以由用户利用编程器写入信息,而且可以对其内容进行多次改写。

EPROM 出厂时,存储内容为全“1”,用户可以根据需要将其中某些记忆单元改为“0”。当需要更新存储内容时可以将原存储内容擦除(恢复全“1”),以便再写入新的内容。

EPROM 又可分为两种:紫外线擦除(UVEPROM)和电擦除(EEPROM)。

UVEPROM 需用紫外线灯制作的擦抹器照射存储器芯片上的透明窗口,使芯片中原存内容被擦除。由于是用紫外线灯进行擦除,所以只能对整个芯片擦除,而不能对芯片中个别需要改写的存储单元单独擦除。另外,为了防止存储的信息受日光中紫外线成分的作用而

缓慢丢失，在UVEPROM芯片写入完成后，必须用不透明的黑纸将芯片上的透明窗口封住。

EEPROM是采用电气方法来进行擦除的，在联机条件下既可以用字擦除方式擦除，也可以用数据块擦除方式擦除。以字擦除方式操作时，能够只擦除被选中的那个存储单元的内容；在数据块擦除方式操作时，可擦除数据块内所有单元的内容。

EPROM虽然既可读，又可写，但它却不能取代RAM。原因如下：

① EPROM的编程次数(寿命)是有限的。

② 写入时间过长，即使对于EEPROM，擦除一个字节大约需要10ms，写入一个字节大约需要10μs，比SRAM或DRAM的时间长100～1000倍。

(4) 闪速存储器

闪速存储器(Flash Memory，简称闪存)是一种允许在操作中被多次擦除或重写的只读存储器，它的主要特点是既可在不加电的情况下长期保存信息，又能在线进行快速擦除与重写，兼备了EEPROM和RAM的优点。

闪速存储器有NOR型和NAND型两种。NOR Flash需要很长的时间进行擦写，允许随机存取存储器上的任何区域，读取数据的方式与从RAM读取数据很相近，这使得它非常适合取代老式的ROM芯片。NAND Flash具有较快的擦写时间，但必须以区块为单位进行读取。

目前，大多数微型计算机的主板采用闪速存储器来存储BIOS程序。由于BIOS的数据和程序非常重要，不允许修改，故早期主板BIOS芯片多采用PROM或EPROM。闪速存储器除了具有ROM的一般特性外，还有低电压改写的特点，便于用户自动升级BIOS。

2. ROM芯片

ROM中使用最多的是可擦除可编程ROM(EPROM)。各种EPROM芯片的外引脚主要有：

- 地址线：A_i。
- 数据线：D_i。
- 片选线：$\overline{CS}$(或$\overline{CE}$)。
- 编程线：$\overline{PGM}$。
- 电源线：V_{CC}——+5V，工作电源；V_{PP}——编程电源；GND——地。V_{PP}平时接+5V，编程写入时，需接高于V_{CC}若干倍的编程电压。

5.3.5 半导体存储器的封装

1. DIP存储芯片

过去，一般存储芯片都是双列直插封装(Dual In-line Package，DIP)的。这种内存芯片必须焊接在主板上才能使用，一旦某一块芯片坏了，必须焊下来才能更换。

DIP芯片的容量一般不可能很大，如64K×1或256K×1的芯片，表示每个芯片具有64K或256K个记忆单元，若要存储256KB的信息，则需要8个256K×1的芯片(非奇偶校验)或9个这样的芯片(奇偶校验)。

2. 内存条

目前,厂家广泛地使用单列直插存储模块(Single In-line Memory Module,SIMM)、双列直插存储模块(Dual In-line Memory Module,DIMM)以及Rambus直插存储模块(Rambus In-line Memory Module,RIMM),这些就是通常所说的内存条。内存条实际上是一条焊有多片存储芯片的印刷电路板,插在主板内存插槽中(不同的内存条必须安装在与其对应的专用插槽上),这样内存条就可以随意拆卸了。

SIMM有30线和72线两种。30线的SIMM诞生于286时代,数据线的宽度只有8位(部分另加有1位校验位),需要用4条SIMM组成一组来构成具有某种容量和32位数据宽度的主存储器。72线的SIMM,数据线的宽度有32位(非奇偶校验)或36位(奇偶校验),每一条就可以构成具有某种容量和32位数据宽度的主存储器。

DIMM有多种类型:标准的DIMM、DDR DIMM、DDR2 DIMM、DDR3 DIMM和DDR4 DIMM。标准的DIMM每面84线,双面共有84线×2=168线,故而常称为168线内存条。而DDR每面92线,双面共有184线。DDR2和DDR3每面都是120线,双面共有240线,但缺口的位置有所不同。DDR4的双面共有284线。所有DIMM的数据线宽度都是64位(非奇偶校验)或72位(奇偶校验),所以在现代PC中,只需一条DIMM就可构成具有某种容量和64位数据宽度的主存储器。

为了满足便携式计算机对内存尺寸的要求,SO-DIMM(Small Outline Dual In-line Memory Module)应运而生。SO-DIMM中文含意为“小外形双列直插存储模块”,它的外形尺寸大致是正常DIMM尺寸的一半。SO-DIMM具有72线(支持32位数据传输)以及144线、200线和260线(支持64位数据传输)等。

RIMM也是双面的,目前只有一种RIMM,它有184线。一个通道通常有3个RIMM插槽,所有RIMM插槽必须全部插满,如有空余则要用专用的Rambus终结器填满。

大多数主板不允许用户把不同容量的内存条混用,用户应在满足主存容量的同时,使内存条的数目尽可能少,这将为进一步扩充主存容量留下余地。

5.4 主存储器的连接与控制

由于存储芯片的容量是有限的,主存储器往往是要由一定数量的芯片构成的。而由若干芯片构成的主存还需要与CPU连接,才能在CPU的正确控制下完成读写操作。

5.4.1 主存容量的扩展

要组成一个主存,首先要考虑选片的问题,然后就是如何把芯片连接起来的问题。根据存储器所要求的容量和选定的存储芯片的容量,就可以计算出总的芯片数,即

$$总片数=\frac{总容量}{容量/片}$$

例如,存储器容量为8K×8,若选用1K×4的存储芯片,则需要:

$$\frac{8K\times 8}{1K\times 4}=8\times 2\ 片=16\ 片$$

将多片组合起来常采用位扩展法、字扩展法、字和位同时扩展法。

1. 位扩展

位扩展是指只在位数方向扩展(加大字长),而芯片的字数和存储器的字数是一致的。位扩展的连接方式是将各存储芯片的地址线、片选线和读写线相应地并联起来,而将各芯片的数据线单独列出。

如用 64K×1 的 SRAM 芯片组成 64K×8 的存储器,所需芯片数为:

$$\frac{64K\times 8}{64K\times 1}=8\ 片$$

在这种情况下,CPU 将提供 16 根地址线($2^{16}=65\ 536$)、8 根数据线与存储器相连;而存储芯片仅有 16 根地址线、1 根数据线。具体的连接方法是:8 个芯片的地址线 $A_{15}\sim A_0$ 分别连在一起,各芯片的片选信号$\overline{CS}$以及读写控制信号$\overline{WE}$也都分别连到一起,只有数据线 $D_7\sim D_0$ 各自独立,每片代表一位,如图 5-16 所示。

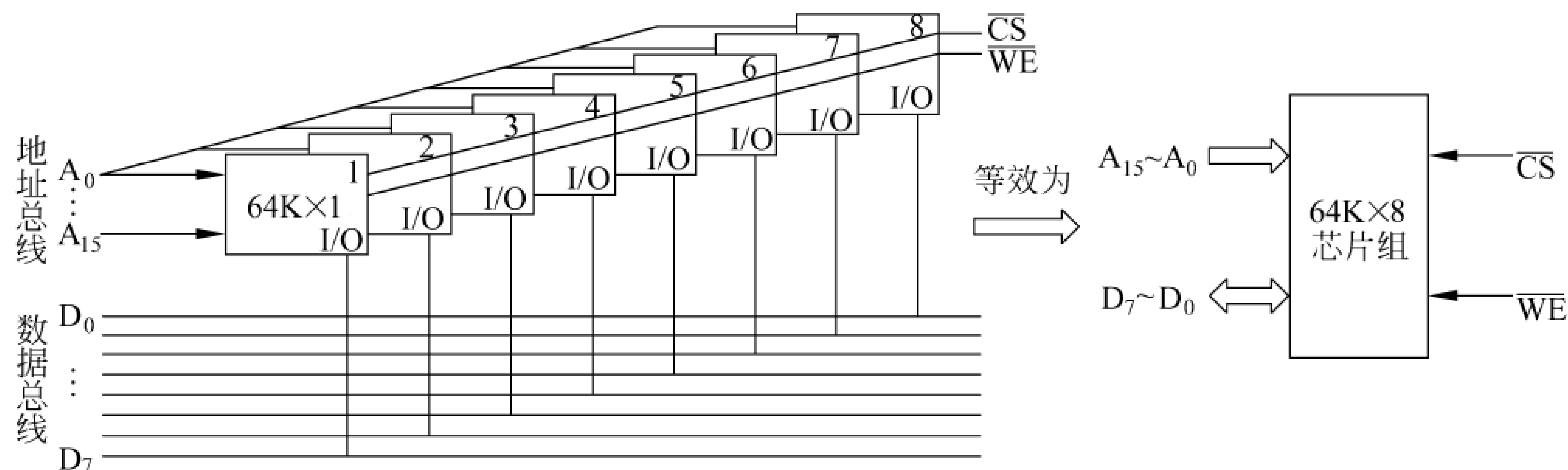


图 5-16 位扩展连接举例

当 CPU 访问该存储器时,其发出的地址和控制信号同时传给 8 个芯片,选中每个芯片的同一单元,相应单元的内容被同时读至数据总线的各位,或将数据总线上的内容分别同时写入相应单元。

2. 字扩展

字扩展是指仅在字数方向扩展,而位数不变。字扩展将芯片的地址线、数据线、读写线并联,由片选信号来区分各个芯片。

如用 16K×8 的 SRAM 组成 64K×8 的存储器,所需芯片数为:

$$\frac{64K\times 8}{16K\times 8}=4\ 片$$

在这种情况下,CPU 将提供 16 根地址线、8 根数据线与存储器相连;而存储芯片仅有 14 根地址线、8 根数据线。4 个芯片的地址线 $A_{13}\sim A_0$、数据线 $D_7\sim D_0$ 及读写控制信号$\overline{WE}$都是同名信号并联在一起;CPU 的高位地址线 A_{15}、A_{14} 经过一个地址译码器产生 4 个片选信号$\overline{CS_i}$,分别选中 4 个芯片中的一个,如图 5-17 所示。

$A_{15}A_{14}=00$,选中第一片;$A_{15}A_{14}=01$,选中第二片……

在同一时间内 4 个芯片中只能有一个芯片被选中。4 个芯片的地址分配如下:

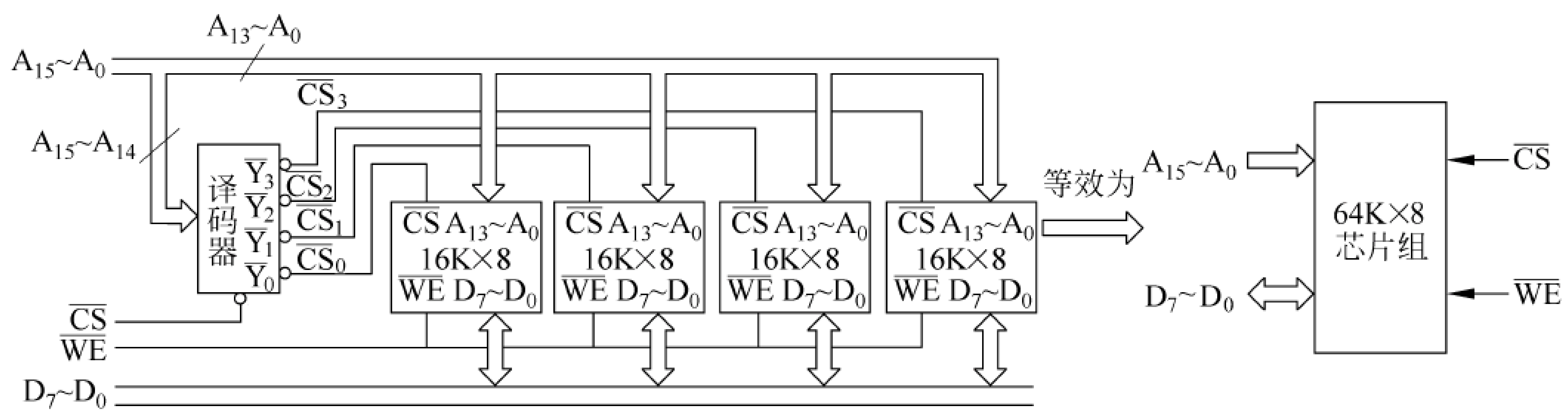


图 5-17　字扩展连接举例

第一片	最低地址	**00**00 0000 0000 0000B	0000H
	最高地址	**00**11 1111 1111 1111B	3FFFH
第二片	最低地址	**01**00 0000 0000 0000B	4000H
	最高地址	**01**11 1111 1111 1111B	7FFFH
第三片	最低地址	**10**00 0000 0000 0000B	8000H
	最高地址	**10**11 1111 1111 1111B	BFFFH
第四片	最低地址	**11**00 0000 0000 0000B	C000H
	最高地址	**11**11 1111 1111 1111B	FFFFH

3. 字和位同时扩展

当构成一个容量较大的存储器时，往往需要在字数方向和位数方向上同时扩展，这将是前两种扩展的组合，实现起来也是很容易的。

图 5-18 表示用 8 片 16K×4 的 SRAM 芯片组成 64K×8 存储器的连接图。

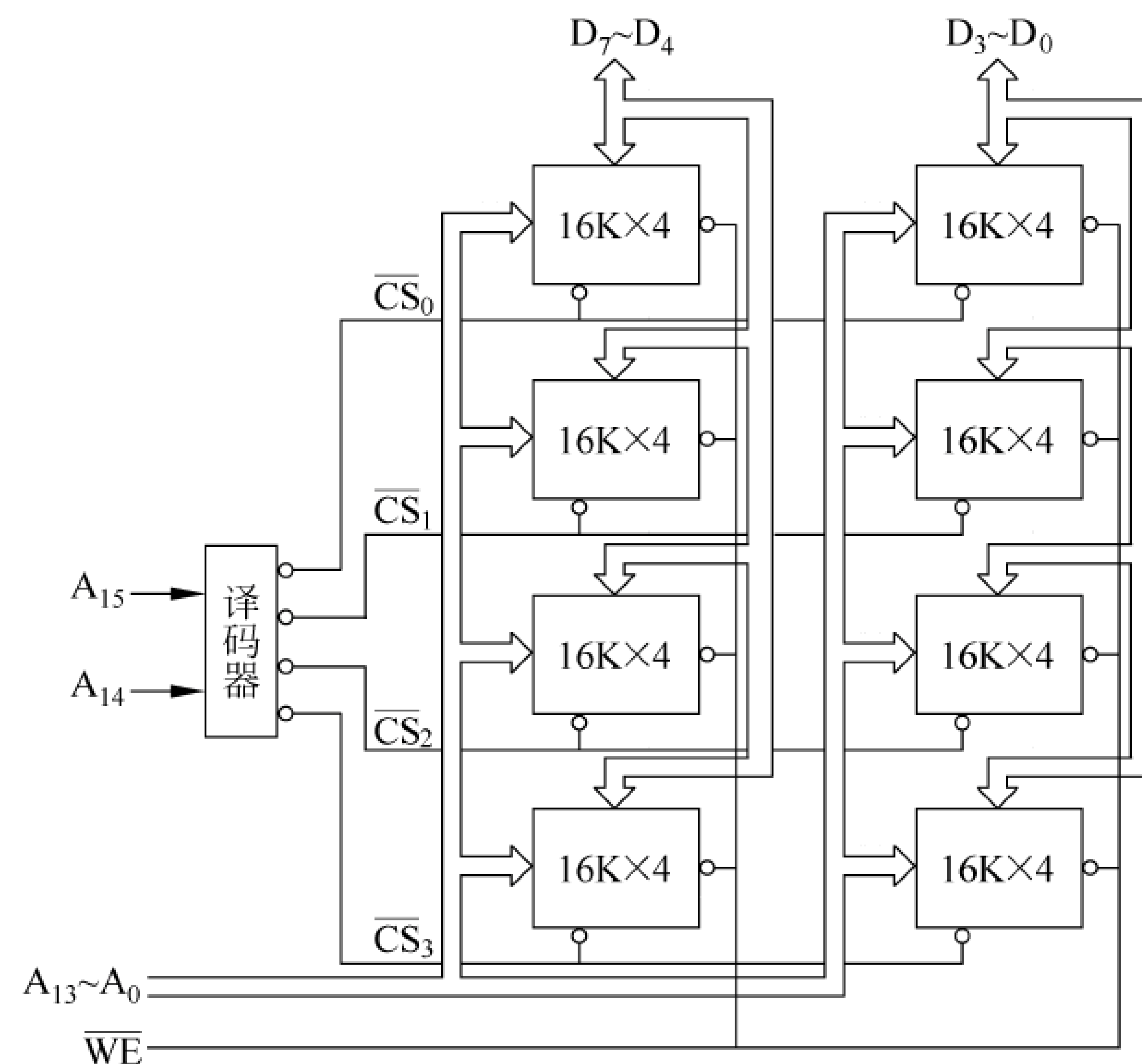


图 5-18　字和位同时扩展连接举例

不同的扩展方法可以得到不同容量的存储器。在选择存储芯片时，一般应尽可能使用集成度高的存储芯片来满足总的存储容量的要求，这样可减少成本，还可减轻系统负载，缩小存储器模块的尺寸。

5.4.2 存储芯片的地址分配和片选

CPU 与存储器连接时，特别是在扩展存储容量的场合下，主存的地址分配是一个重要的问题。确定地址分配后，又有一个存储芯片的片选信号的产生问题。

CPU 要实现对存储单元的访问，首先要选择存储芯片，即进行片选；然后再从选中的芯片中依地址码选择出相应的存储单元，以进行数据的存取，这称为字选。片内的字选是由 CPU 送出的 N 条低位地址线完成的，地址线直接接到所有存储芯片的地址输入端（N 由片内存储容量 2^N 决定）。而存储芯片的片选信号则大多是通过高位地址译码或直接连接产生的。

片选信号的产生可细分为线选法、全译码法和部分译码法。

1. 线选法

线选法就是用除片内寻址外的高位地址线直接（或经反相器）分别接至各个存储芯片的片选端，当某地址线信息为“0”时，就选中与之对应的存储芯片。

注意：这些片选地址线每次寻址时只能有一位有效，不允许同时有多位有效，这样才能保证每次只选中一个芯片（或组）。

假设 4 片 2K×8 用线选法构成 8K×8 存储器，各芯片的地址分配如表 5-3 所示。

表 5-3 线选法的地址分配

芯　　片	A_{14}～A_{11}	A_{10}～A_0	地址范围（空间）
0#	1 1 1 0	00…0 ⋮ 11…1	7000～77FFH
1#	1 1 0 1	00…0 ⋮ 11…1	6800～6FFFH
2#	1 0 1 1	00…0 ⋮ 11…1	5800～5FFFH
3#	0 1 1 1	00…0 ⋮ 11…1	3800～3FFFH

线选法的优点是不需要地址译码器，线路简单，选择芯片无须外加逻辑电路，但仅适用于连接存储芯片较少的场合。同时，线选法不能充分利用系统的存储器空间，且把地址空间分成相互隔离的区域，给编程带来了一定的困难。

2. 全译码法

全译码法将除片内寻址外的全部高位地址线都作为地址译码器的输入，译码器的输出

作为各芯片的片选信号,将它们分别接到存储芯片的片选端,以实现对存储芯片的选择。

全译码法的优点是每片(或组)芯片的地址范围是唯一确定的,而且是连续的,也便于扩展,不会产生地址重叠的存储区,但全译码法对译码电路要求较高。

例如,CPU 的地址总线有 20 位,现用 4 片 2K×8 的存储芯片组成一个 8K×8 的存储器。全译码法要求除去片内寻址用到的 11 位地址线外,高 9 位地址 A_{19}～A_{11} 都要参与译码。各芯片的地址分配如表 5-4 所示。

表 5-4　全译码法的地址分配

芯　　片	A_{19}～A_{13}	$A_{12}A_{11}$	A_{10}～A_0	地址范围(空间)
0#	0…0	0 0	00…0 ⋮ 11…1	00000～007FFH
1#	0…0	0 1	00…0 ⋮ 11…1	00800～00FFFH
2#	0…0	1 0	00…0 ⋮ 11…1	01000～017FFH
3#	0…0	1 1	00…0 ⋮ 11…1	01800～01FFFH

3. 部分译码法

所谓部分译码就是用除片内寻址外的高位地址的一部分来译码产生片选信号。如用 4 片 2K×8 的存储芯片组成 8K×8 存储器,需要 4 个片选信号,因此只需要用两位地址线来译码产生。

由于寻址 8K×8 存储器时未用到高位地址 A_{19}～A_{13},所以无论 A_{19}～A_{13} 取何值,只要 $A_{12}=A_{11}=0$,则选中第一片;只要 $A_{12}=0$,$A_{11}=1$,则选中第二片;……也就是说,8K RAM 中的任一个存储单元,都对应有 $2^{(20-13)}=2^7$ 个地址,这种一个存储单元出现多个地址的现象称地址重叠。

从地址分布来看,这 8KB 存储器实际上占用了 CPU 全部的空间(1MB)。每片 2K×8 的存储芯片有 $\frac{1}{4}$M=256K 的地址重叠区,如图 5-19 所示。

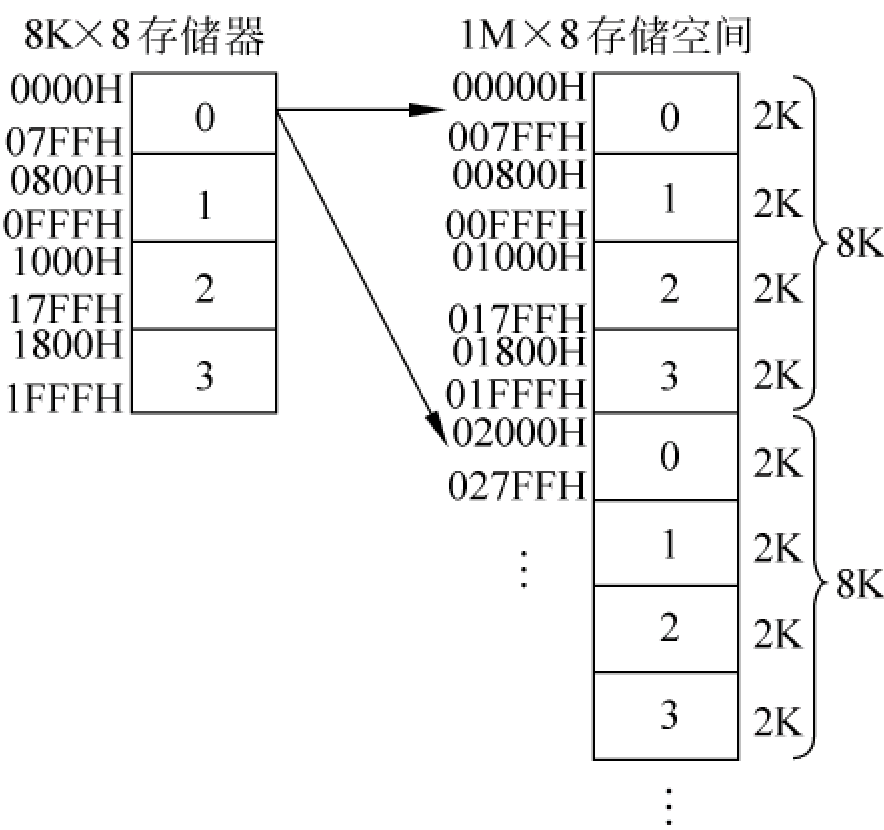


图 5-19　地址重叠区示意图

令未用到的高位地址全为 0,这样确定的存储器地址称为基本地址,本例中 8K×8 存储器的基本地址即 00000H～01FFFH。部分译码法较全译码法简单,但存在地址重叠区。

5.4.3 主存储器和CPU的连接

在讨论了主存的结构之后，进一步了解主存和CPU之间的连接是十分必要的。

1. 主存和CPU之间的硬连接

主存与CPU的硬连接有3组连线：地址总线(AB)、数据总线(DB)和控制总线(CB)，如图5-20所示。此时，把主存看作一个黑盒子，存储器地址寄存器(MAR)和存储器数据寄存器(MDR)是主存和CPU之间的接口。MAR可以接受来自程序计数器(PC)的指令地址或来自地址形成部件的操作数地址，以确定要访问的单元。MDR是向主存写入数据或从主存读出数据的缓冲部件。MAR和MDR从功能上看属于主存，但在小型计算机、微型计算机中常放在CPU内。

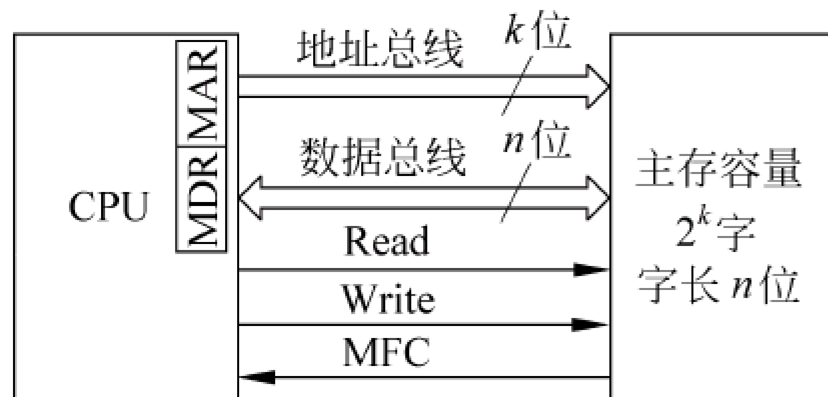


图5-20 主存和CPU的硬连接

2. CPU对主存的基本操作

前面所说的CPU与主存的硬连接是两个部件之间联系的物理基础。而两个部件之间还有软连接，即CPU向主存发出的读或写命令，这才是两个部件之间有效工作的关键。

CPU对主存进行读写操作时，首先CPU在地址总线上给出地址信号，然后发出相应的读或写命令，并在数据总线上交换信息。读写的基本操作如下。

(1) 读

读操作是指从CPU送来的地址所指定的存储单元中取出信息，再送给CPU，其操作过程是：

① 地址→MAR→AB　　CPU将地址信号送至地址总线。
② Read　　CPU发读命令。
③ Wait for MFC　　等待存储器工作完成信号。
④ M(MAR)→DB→MDR　　读出信息经数据总线送至CPU。

(2) 写

写操作是指将要写入的信息存入CPU所指定的存储单元中，其操作过程是：

① 地址→MAR→AB　　CPU将地址信号送至地址总线。
② 数据→MDR→DB　　CPU将要写入的数据送至数据总线。
③ Write　　CPU发写命令。
④ Wait for MFC　　等待存储器工作完成信号。

由于CPU和主存的速度存在着差距，所以两者之间的速度匹配是很关键的。通常有两种匹配方式：同步存储器读取和异步存储器读取。上面给出的读写基本操作是以异步存储器读取来考虑的，CPU和主存之间没有统一的时钟，由主存工作完成信号(MFC)通知CPU“主存工作已完成”。对于读操作，若MFC＝1，说明信息已经读出；对于写操作，若MFC＝1，说明数据已写入相应的存储单元。

对于同步存储器读取，CPU和主存采用统一时钟，同步工作，因为主存速度较慢，所以

CPU与之配合必须放慢速度。在这种存储器中,不需要主存工作完成信号。

5.4.4 主存的校验

计算机在运行过程中,主存要与CPU频繁地交换数据。为了检测和校正在存储过程中的错误,主存中常设置有差错校验电路。

1. 主存的奇偶校验

最简单的主存检验方法是奇偶校验,有关奇偶校验的概念已经在第2章进行了讨论。在微机中通常采用奇校验,即每个存储单元中共存储9位信息(其中8位数据,1位奇偶校验位),信息中“1”的个数总是奇数。

当向主存写入数据时,奇偶校验电路首先会对一个字节的数据计算出奇偶校验位的值,然后再把所有的9位值一起送到主存中。

读出数据时,某一存储单元的9位数据被同时读出,当9位数据里“1”的个数为奇数时,表示读出的9位数据正确(当然不排除有2位同时出错的可能,但其概率极小);当“1”的个数为偶数时,表示读出数据出错,向CPU发出不可屏蔽中断,使系统停机并显示奇偶检验出错的信息。

2. 错误检验与校正(ECC)

虽然奇偶校验主存仍在使用,但它的继承者“错误校验与校正(Error Checking and Correcting,ECC)”已经广泛取代了它,ECC不仅能检测错误还能在不打扰计算机工作的情况下改正错误,这对于网络服务器这样不允许随便停机的关键任务是至关重要的。最常用的ECC就是第2章中提到的汉明码校验,可对已访问的数据字段进行单位错误的检测和修复,而对双位错误只能检测不能修复。

ECC主存用一组附加数据位来存储一个特殊码,被称为“校验和”。对于每个二进制字都有相应的ECC码。产生ECC码所需的位数取决于系统所用的二进制字长。例如,32位字要求有7位ECC码,此时ECC的开销大于奇偶校验的开销;64位字要求有8位ECC码,此时ECC和奇偶校验的开销是一样的。

ECC在存储器写操作时需要存储器控制器计算校验位,当从主存中读取数据时,将取到的实际数据和它的ECC码快速比较。如果匹配,则实际数据被传给CPU;如果不匹配,则ECC码的结构能够将出错的一位鉴别出来,然后改正错误,再将数据传给CPU。

注意:此时主存中的出错位并没有改变,如果又要读取这个数据,需要再一次校正错误。

大多数存储器的错误具有单位出错的特征,能够被ECC纠正过来,这种容错技术提高了系统的可靠性和可用性。基于ECC的系统是服务器、工作站和重要应用的最佳选择。

现代的PC中主存的容错能力被分为基本的三级:

① 无奇偶检验。

② 奇偶检验。

③ ECC。

无奇偶校验的主存根本就没有容错能力。它们之所以被使用,仅仅是因为其价格最低,

且无奇偶校验主存的控制部件相对简单。

5.4.5 PC系列微型计算机的存储器接口

8088、8086、80386和Pentium微处理器的外部数据总线分别是8位、16位、32位和64位，下面介绍它们与主存的接口。

1. 8位存储器接口

如果数据总线为8位(如微机系统中的PC总线)，而主存按字节编址，则匹配关系比较简单。对于8位的微处理器，典型的时序安排是占用4个CPU时钟周期，称为$T_1 \sim T_4$，构成一个总线周期。对于微型计算机来说，存储器就接在总线上，故总线周期就等于存取周期，一个总线周期可读写8位。

8位的微处理器8088提供$\overline{RD}$(读选通)、$\overline{WR}$(写选通)和IO/$\overline{M}$(I/O或存储器控制)等控制信号(最小模式)去控制存储器系统，或者提供IO/$\overline{M}$与$\overline{RD}$一起产生的$\overline{MRDC}$(存储器读命令)、IO/$\overline{M}$与$\overline{WR}$一起产生的$\overline{MWTC}$(存储器写命令)等控制信号(最大模式)去控制存储器系统。

2. 16位存储器接口

对于16位的微处理器8086(或80286)，在一个总线周期内最多可读写两个字节，即从偶地址开始的字(规则字)。同时，读写这个偶地址单元和随后的奇地址单元，用低8位数据总线传送偶地址单元的数据，用高8位数据总线传送奇地址单元的数据。如果读写的是非规则字，即是从奇地址开始的字，则需要安排两个存取周期才能实现。

为了实现这样的传送，需要将存储器分为两个存储体，如图5-21所示。一个存储体的地址均为偶数，称为偶地址(低字节)存储体，它与低8位数据总线相连；另一个存储体的地址均为奇数，称为奇地址(高字节)存储体，与高8位数据总线相连。

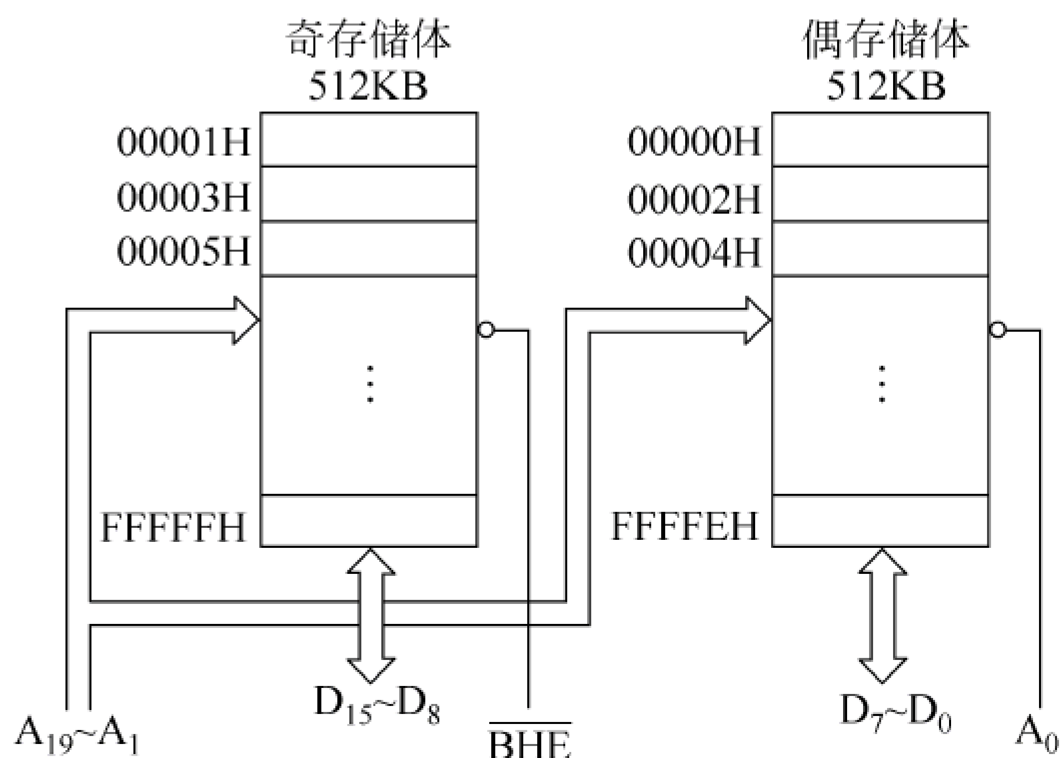


图5-21　8086的存储器组织

8086微处理器的地址线$A_{19} \sim A_1$同时送至两个存储体，$\overline{BHE}$(高位存储体)和最低位地址线A_0用来选择一个或两个存储体进行数据传送。$\overline{BHE}$和A_0的选择如表5-5所示。

表 5-5　$\overline{BHE}$和 A_0 的选择表

$\overline{BHE}$	A_0	特　　征
0	0	全字(规则字)传送
0	1	在数据总线高 8 位上进行字节传送
1	0	在数据总线低 8 位上进行字节传送
1	1	备用

8086 和主存之间可以传送一个字节(8 位)数据,也可以传送一个字(16 位)数据。任何两个连续的字节都可以作为一个字来访问,地址值较低的字节是低位有效字节,地址值较高的字节是高位有效字节。

图 5-22 给出了各种信息的传送方法:图 5-22(a)为偶地址字节传送,图 5-22(b)为奇地址字节传送;图 5-22(c)为偶地址字传送,图 5-22(d)和图 5-22(e)为奇地址字传送。从图中可以看出,对于规则字(边界对齐的偶地址字)进行读写,仅需一个存取周期;而对于非规则字(边界未对齐的奇地址字)进行读写,就需要两个存取周期,而且每次都应忽略掉不需要的半个字。

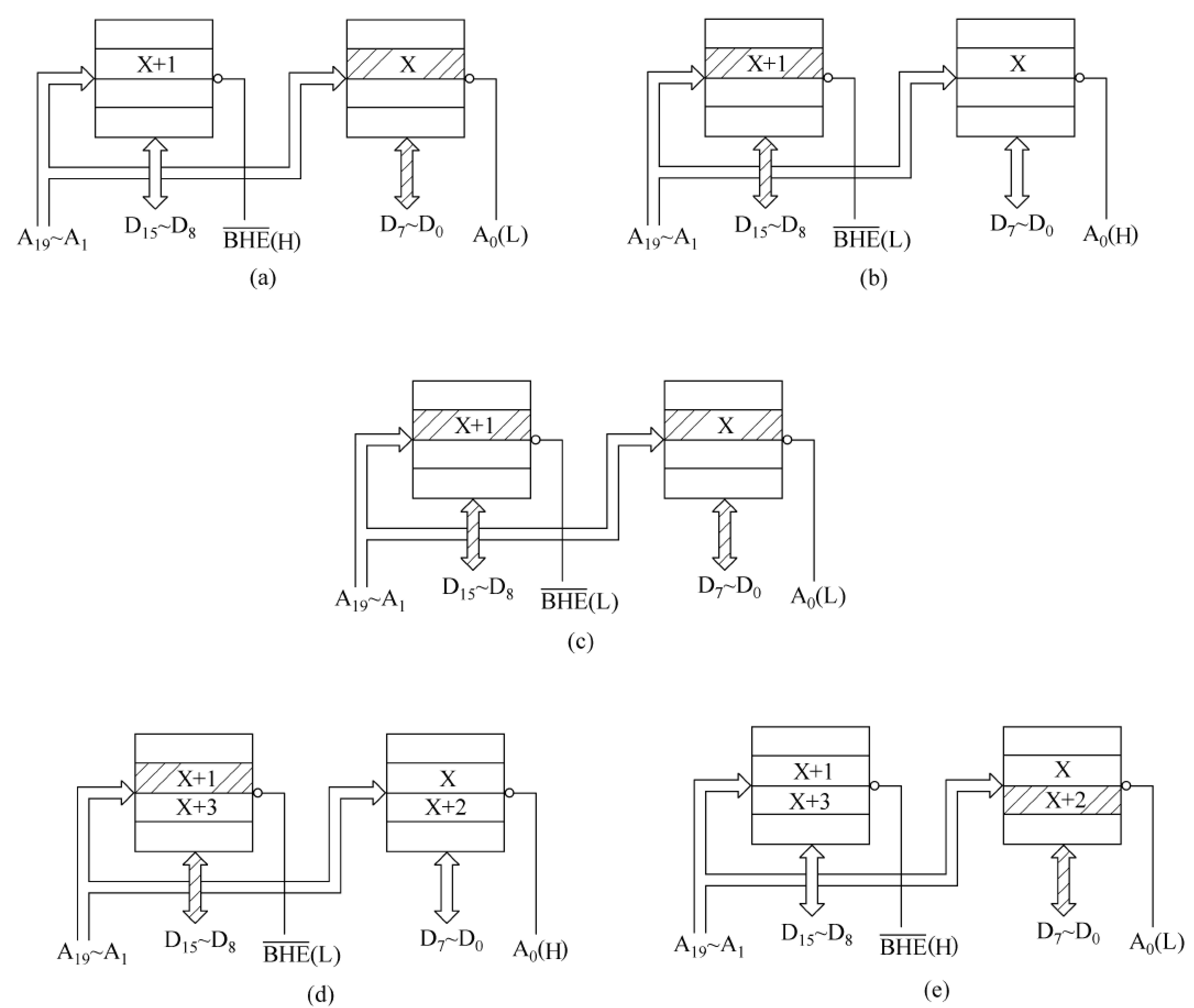


图 5-22　各种信息的传送方法

3. 32 位存储器接口

由于 80386/80486 微处理器要保持与 8086 等微处理器兼容，这就要求在进行存储器系统设计时必须满足单字节、双字节和四字节等不同访问。为了实现 8 位、16 位和 32 位数据的访问，80386/80486 微处理器设有 4 个引脚$\overline{BE_3}$～$\overline{BE_0}$，以控制不同数据的访问。$\overline{BE_3}$～$\overline{BE_0}$ 由 CPU 根据指令的类型产生，其作用如表 5-6 所示。

表 5-6　$\overline{BE_3}$～$\overline{BE_0}$ 功能表

字节允许				要访问的数据位				自动重复
$\overline{BE_3}$	$\overline{BE_2}$	$\overline{BE_1}$	$\overline{BE_0}$	D_{31}～D_{24}	D_{23}～D_{16}	D_{15}～D_8	D_7～D_0	
1	1	1	0	—	—	—	D_7～D_0	N
1	1	0	1	—	—	D_{15}～D_8	—	N
1	0	1	1	—	D_{23}～D_{16}	—	D_{23}～D_{16}	Y
0	1	1	1	D_{31}～D_{24}	—	D_{31}～D_{24}	—	Y
1	1	0	0	—	—	D_{15}～D_8	D_7～D_0	N
1	0	0	1	—	D_{23}～D_{16}	D_{15}～D_8	—	N
0	0	1	1	D_{31}～D_{24}	D_{23}～D_{16}	D_{31}～D_{24}	D_{23}～D_{16}	Y
1	0	0	0	—	D_{23}～D_{16}	D_{15}～D_8	D_7～D_0	N
0	0	0	1	D_{31}～D_{24}	D_{23}～D_{16}	D_{15}～D_8	—	N
0	0	0	0	D_{31}～D_{24}	D_{23}～D_{16}	D_{15}～D_8	D_7～D_0	N

从表 5-6 中可以看出，在 8 位和 16 位数据传送中，当微处理器写入高字节或高 16 位数据时，该数据将在低字节或低 16 位数据线上重复输出。其目的是为了加快数据传送的速度，但是是否能够写入低字节或低 16 位单元，则由相应的$\overline{BE_i}$ 决定。

32 位微处理器的存储器组织如图 5-23 所示。80386/80486 微处理器有 32 位地址线，但是直接输入 A_{31}～A_2，低两位 A_1～A_0 由内部编码产生$\overline{BE_3}$～$\overline{BE_0}$，以选择不同字节。主存由 4 个存储体组成，每个存储体的存储空间可达 1GB。如果要访问一个 32 位数，那么 4 个存储体都被选中；若要访问一个 16 位数，则有两个存储体（通常是$\overline{BE_3}$ 和$\overline{BE_2}$ 或者$\overline{BE_1}$ 和$\overline{BE_0}$）被选中；若访问的是 8 位数，只有一个存储体被选中。

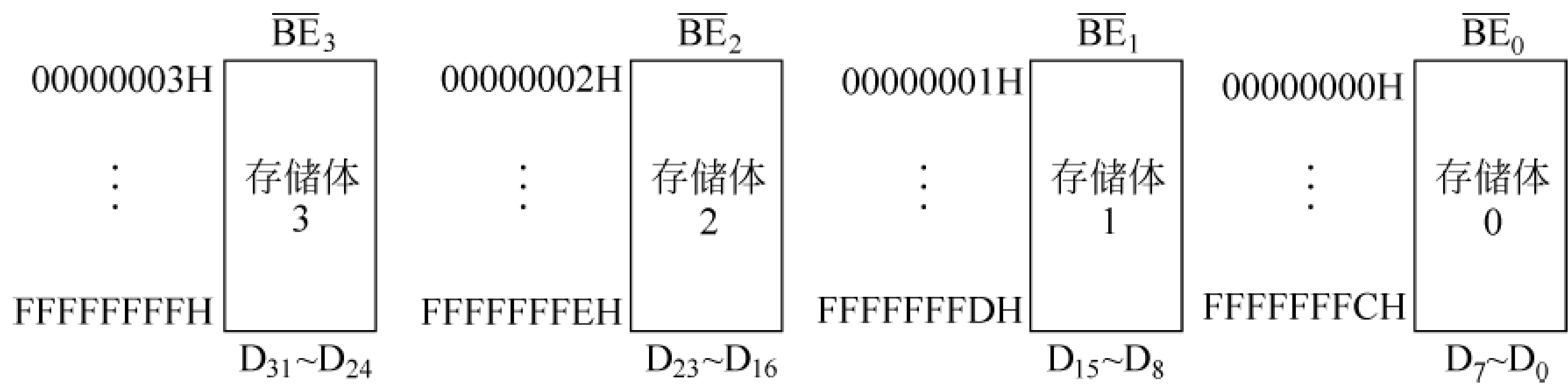


图 5-23　32 位微处理器的存储器组织

4. 64 位存储器接口

64 位存储器系统由 8 个存储体组成，每个存储体的存储空间为 512MB（Pentium）或

8GB(Pentium Pro),存储体选择通过选择信号$\overline{BE_7}$~$\overline{BE_0}$实现。如果要传送一个64位数,那么8个存储体都被选中;如果要传送一个32位数,那么4个存储体被选中;若要传送一个16位数,则有两个存储体被选中;若传送的是8位数,只有一个存储体被选中。

64位存储器组织与前述32位存储器组织相似,在此不再重复。图5-24给出了Pentium微处理器的地址总线与64、32、16和8位存储器的接口示意图。

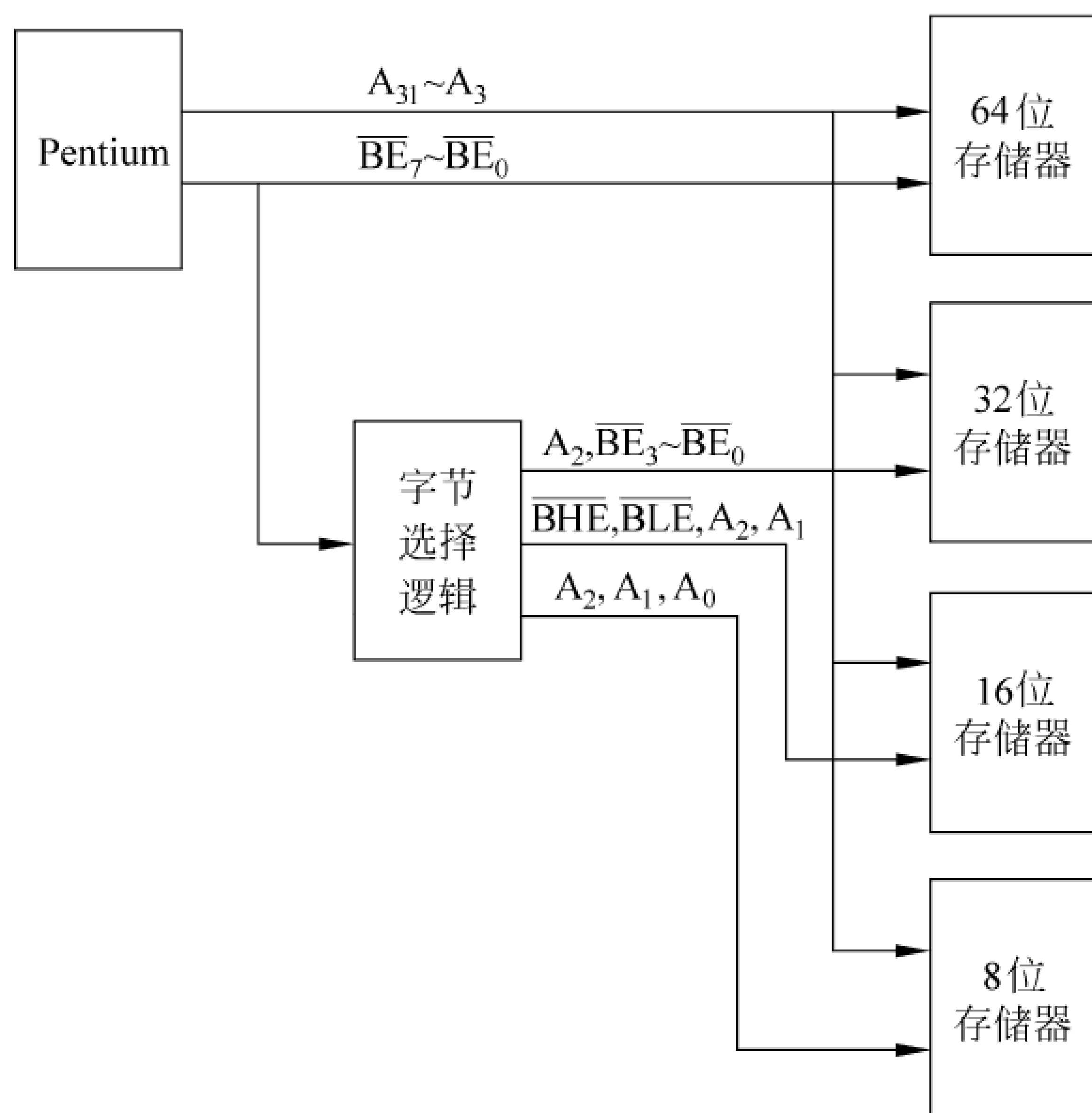


图5-24 Pentium微处理器的地址总线与64、32、16和8位存储器的接口

5.5 提高主存读写速度的技术

近几年来,主存技术一直在不断地发展,从最早使用的DRAM到后来的FPM DRAM、EDO DRAM、SDRAM、DDR SDRAM、DDR2 SDRAM、DDR3 SDRAM、DDR4 SDRAM和RDRAM,出现了各种主存控制与访问技术,它们的共同特点是使主存的读写速度有了很大的提高。

5.5.1 主存与CPU速度的匹配

过去,主存的速度通常以纳秒(ns)表示,而CPU速度总是被表示为兆赫兹(MHz),最近一些更快更新的主存也用MHz来表示速度。

如果主存总线的速度与CPU总线速度相等,那么主存的性能将是最优的。然而通常主存的速度落后于CPU的速度,以个人计算机(PC)为例,在1998年以前,DRAM的存取时间为60ns或更大,这相当于16.7MHz或更慢的速度,而当时CPU的速度已达到300MHz或更高的速度,两者之间存在着很大的差距,这就是为什么需要高速缓冲存储器(Cache)的原因。

当1GHz CPU要从133MHz主存读多个字节的数据时会出现大量的等待状态,所谓等

待状态就是处理器在等待数据就绪之前必须执行的一个额外"什么都不做"的周期。由于主存周期为7.5ns,CPU周期为1ns,CPU需要执行6个等待周期,然后数据才会在第七个周期准备好。增加等待周期实际上是将CPU速度减慢至主存速度。为了减少所需的等待周期数,许多系统开始引入新型的存储芯片,这些存储芯片在存储器总线的性能已与CPU总线的性能相差无几。

5.5.2 FPM DRAM

传统的DRAM是通过分页技术进行访问的,在存取数据时,需要分别输入一个行地址和一个列地址,这会耗费时间。快速页模式随机存储器(Fast Page Mode DRAM,FPM DRAM)是传统DRAM的改进型产品,通过保持行地址不变而只改变列地址,可以对给定行的所有数据进行更快的访问。FPM DRAM的速度之所以能提高是基于这样一个事实——计算机中大量的数据是连续存放的。例如,若一个数据与前一个数据的行地址相同,主存控制器就不必再传一次行地址,只要再传一个列地址就可以了。这种触发行地址后连续输出列地址的方式能用较少的时钟周期读较多的数据,即存取同一"页"数据的速度与效率就大大提高了(行地址不变时,列地址可寻址的空间称为一"页",一页通常为1024字节的整数倍)。

FPM DRAM还支持突发模式访问,所谓突发模式是指对一个给定的访问在建立行和列地址之后,可以访问后面3个相邻的地址,而不需要额外的延迟和等待状态。一个突发访问通常限制为4次正常访问。为了描述这个过程,经常以每次访问的周期数表示计时。一个标准DRAM的典型突发模式访问表示为x-y-y-y,x是第一次访问的时间(延迟加上周期数),y表示后面每个连续访问所需的周期数。标准的FPM DRAM可获得5-3-3-3的突发模式周期。

显然,FPMD RAM的这种工作方式需要存储芯片和主存控制器共同配合才能完成。在Pentium主板上,主存控制器被制作在主板的芯片组中。随着技术的成熟,FPM DRAM的访问时间也在不断缩短,从120ns缩短到60ns。

FPM DRAM内存条主要采用72线的SIMM封装,其存取速度一般为60～100ns。

5.5.3 EDO DRAM

扩展数据输出DRAM(Extended Data Output DRAM,EDO DRAM)是在FPM DRAM基础上加以改进的存储器控制技术。传统的DRAM和FPM DRAM在存取每一数据时,输入行地址和列地址后必须等待电路稳定,然后才能有效地读写数据,而下一个地址必须等待这次读写周期完成才能输出。而EDO输出数据在整个CAS周期都是有效的(包括预充电时间在内),EDO不必等待当前的读写周期完成即可启动下一个读写周期,即可以在输出一个数据的过程中准备下一个数据的输出。EDO DRAM采用一种特殊的主存读出控制逻辑,在读写一个存储单元时同时启动下一个(连续)存储单元的读写周期,从而节省了重选地址的时间,提高了读写速度。

EDO DRAM可获得5-2-2-2的突发模式周期,若进行4个主存传输,需要总共11个系统周期,而FPM DRAM的突发模式周期为5-3-3-3,总共需要14个周期。与FPM DRAM相比,EDO DRAM的性能改善了22%,而其制造成本与FPM DRAM相近。

FPM和EDO两者的芯片制作技术其实是相同的,不同的是EDO所增加的机制必须在

芯片组的支持下将发送的数据信号的处理时间缩短,以加快系统的整体执行效率。EDO DRAM内存条主要采用72线的SIMM形式封装,也有少部分采用168线的DIMM封装,工作电压为5V,存取时间为50～70ns。

5.5.4 SDRAM

前面介绍的几种DRAM主存都属于"非同步存取的存储器",即它们的工作速度并没有和系统时钟同步,存取数据时,系统必须等待若干时钟周期才能接收和发送数据。例如,EDO DRAM必须等待2个时钟周期,FPM DRAM则必须等待3个时钟周期,这种等待制约了系统的数据传送速率。通常,FPM DRAM和EDO DRAM的速度不能超过66MHz。

同步动态随机存储器(Synchronous DRAM,SDRAM)是一种与主存总线运行同步的DRAM。SDRAM在同步脉冲的控制下工作,取消了主存等待时间,减少了数据传送的延迟时间,因而加快了系统速度。SDRAM仍然是一种DRAM,起始延迟仍然不变,但总的周期时间比FPM或EDO快得多。SDRAM突发模式可达到5-1-1-1,即进行4个主存传输,仅需8个周期,比EDO快将近20%。

SDRAM的基本原理是将CPU和RAM通过一个相同的时钟锁在一起,使得RAM和CPU能够共享一个时钟周期,以相同的速度同步工作。就是说,SDRAM在开始的时候要多花一些时间,但在以后,每1个时钟可以读写1个数据,做到了所有的输入输出信号与系统时钟同步。这已经接近主板上的同步Cache的3-1-1-1水准。一般来说,在系统时钟为66MHz时,SDRAM与EDO DRAM相比,显示不出其优点,但当系统时钟增加到100MHz以上,SDRAM的优点便很明显。

SDRAM采用新的双存储体结构,内含两个交错的存储矩阵,允许两个主存页面同时打开,当CPU从一个存储矩阵访问数据的同时,在主存控制器作用下另一个存储矩阵已准备好读写数据。通过两个存储矩阵的紧密配合,存取效率得到成倍提高。

SDRAM普遍采用168线的DIMM封装,速度通常以MHz来标定,为降低功耗,一般使用3.3V电压。SDRAM支持PC 66/100/133/150等不同的规范,表示其的工作频率分别为66MHz、100MHz、133MHz和150MHz,能与相应的CPU同步运行,可提高整机性能5%～10%。

5.5.5 DDR SDRAM

双数据传输率同步动态随机存储器(Double Data Rate SDRAM,DDR SDRAM)也可以说是SDRAM的升级版本,DDR SDRAM运用了更先进的同步电路,它与SDRAM的主要区别是:DDR SDRAM不仅能在时钟脉冲的上升沿读出数据而且还能在下降沿读出数据,不需要提高时钟频率就能加倍提高SDRAM的速度。

DDR SDRAM的频率可以用工作频率和等效传输频率两种方式表示,工作频率是内存颗粒实际的工作频率(又称核心频率),但是由于DDR可以在脉冲的上升沿和下降沿都传输数据,因此传输数据的等效传输频率是工作频率的两倍。由于外部数据总线的宽度为64位,所以数据传输率(带宽)等于等效传输频率×8。

DDR SDRAM基本上可完全沿用SDRAM现有的生产体系,其生产成本与SDRAM相差不大。DDR内存条的物理大小和标准的DIMM一样,区别仅在于内存条的线数。标准

的SDRAM有168线(两个小缺口),而DDR SDRAM有184线(多出的16个线占用了空间,故只有1个小缺口)。DDR RDRAM可以工作在2.5V的低电压环境下。

DDR SDRAM的标准主要有DDR 200、DDR 266、DDR 333和DDR 400等,分别对应PC1600/PC2100/PC2700/PC3200几种规范,以DDR 266为例,它的工作频率为133MHz,等效传输频率为266MHz(133MHz×2),传输带宽为2.1GB/s(266×8)。

5.5.6 DDR2、DDR3、DDR4和DDR5 SDRAM

在DDR之后,内存的家族中又陆续出现了DDR2、DDR3、DDR4,DDR5的原型也已经面世。

1. DDR2 SDRAM

DDR2(Double Data Rate 2)SDRAM与上一代DDR SDRAM技术标准最大的不同在于,虽然同是采用了在时钟的上升沿和下降沿同时进行数据传输的基本方式,但DDR2 SDRAM却拥有两倍于上一代DDR SDRAM的预读取能力(即4位数据读预取)。换句话说,DDR2 SDRAM每个时钟能够以4倍于外部总线的速度读写数据,即在同样100MHz的工作频率下,DDR的实际频率为200MHz,而DDR2则可以达到400MHz。

目前,已有的DDR2分为DDR2 400、DDR2 533、DDR2 667、DDR2 800等,其核心频率分别为100MHz、133MHz、166MHz和200MHz,等效的数据传输频率分别为400MHz、533MHz、667MHz和800MHz,其对应的传输带宽分别为3.2GB/s、4.3GB/s、5.3GB/s和6.4GB/s,对应PC2 3200、PC2 4300、PC2 5300、PC2 6400几种规范。

DDR2内存条采用240线DIMM,工作电压1.8V,相对于DDR标准的2.5V下降了不少,从而降低了功耗和发热量。

2. DDR3 SDRAM

DDR3 SDRAM可以看作是DDR2的改进版,DDR2的预取设计位数是4位,即DRAM内核的频率只有接口频率的1/4,而DDR3的预取设计位数提升至8位,其DRAM内核的频率达到了接口频率的1/8。同样运行在200MHz核心工作频率下,DDR2的等效传输频率为800MHz,而DDR3的等效传输频率可以达到1600MHz。

依照JEDEC(电子设备工程联合委员会)的标准,DDR3在800~2133MHz下运行,这是DDR2频率的两倍。DDR3 SDRAM分为DDR3 800、DDR3 1066、DDR3 1333、DDR3 1600等,其核心频率仍分别为100MHz、133MHz、166MHz和200MHz,等效的数据传输频率分别为800MHz、1066MHz、1333MHz和1600MHz,其对应的传输带宽分别为6.4GB/s、8.6GB/s、10.6GB/s和12.8GB/s,对应PC3 6400、PC3 8600、PC3 10600、PC3 12800几种规范。

DDR3内存条仍采用240线DIMM,电压有标准版1.5V、节能版1.35V两种,相比DDR2来说可以节约大约16%的电能。目前,DDR3仍然被用作微机的内存条。

3. DDR4 SDRAM

2012年9月底,JEDEC正式公布了DDR4内存标准规范,由于数据预取的增加变得越

来越困难,所以DDR4推出了bank group设计。每个bank group可独立读写数据,使得内部的数据吞吐率大大提升。DDR4架构上采用了8位预取的bank group分组,包括使用2个或4个可选择的bank group分组,如果内存内部设计了2个独立的bank group,相当于每次操作16位数据,变相的把内存预取值提高到16位,如果是4个独立的bank group,相当于每次操作32位数据,变相的把内存预取值提高到32位。

DDR4内存标准规定最低是DDR4 1600,即从1600MHz开始运行,这将是DDR3频率的两倍。例如,DDR4 3200的带宽为25.6GB/s,比DDR3 1866高出了70%。

DDR4内存条的引脚数从DDR3的240线增加至284线,内存条的外观变化明显,金手指变成弯曲状,以保证与内存插槽点有足够的接触面,且中间的凸起部分与内存插槽产生足够的摩擦力以稳定内存。工作电压下降至1.2V、1.1V,甚至有1.05V的超低压节能版。目前,DDR4已成为主流微机的标准配置。

DDR SDRAM内存的发展趋势如表5-7所示。

表5-7 DDR SDRAM内存的发展趋势

频率(MHz)	200	266	333	400	533	666	800	1066	1333	1600	1866	2133	2666	3200	4266
DDR	√	√	√	√											
DDR2				√	√	√	√	√							
DDR3							√	√	√	√	√	√			
DDR4										√	√	√	√	√	√

4. DDR5 SDRAM

目前,JEDEC组织正在制订DDR5内存标准规范,DDR5的原型也已经开始展示。4400MHz对于DDR5来说可能只是起步,预计最终可以达到6400MHz左右,相比目前的DDR4,频率提升了近一倍。DDR5的变化不仅是频率的提高,因为允许加入内部ECC来制造16Gb、32Gb颗粒,单条容量也会大大提升。

根据业内人士预测,DDR5内存预计将于2020年开始面向服务器和数据中心等企业用户供货,面向消费市场的DDR5内存则还需要等待支持DDR5内存的处理器和主板出现,才会正式商用,预计需要等到2022年。

5.5.7 Rambus DRAM

Rambus DRAM(RDRAM),是继SDRAM之后的新型高速动态随机存储器。由美国Rambus公司研发的RDRAM在内部结构上进行了重新设计,并采用了新的信号接口技术,其对外接口也不同于以前的DRAM。该内存规范是Intel公司与Rambus公司共同定制的,旨在创造市面上最高速的内存产品。

使用FPM/EDO或SDRAM的传统主存系统称为宽通道系统,它们的主存通道和处理器的数据总线一样宽。RDRAM却是一种窄通道系统,它一次只传输16位数据(加上两个可选的校验位),但速度却快得多。目前,RDRAM的容量一般为64Mb、72Mb、128Mb、144Mb,组织结构为4M×16位、4M×18位、8M×16位、8M×18位(18位的组织结构允许

进行 ECC 检测)。

RDRAM 依靠其极高的工作频率,通过减少每个周期的数据量来简化操作。RDRAM 的时钟频率可达到 400MHz,由于采用双沿传输,使原有的 400MHz 变为 800MHz。Rambus 结构的带宽视 Rambus 通路的个数而定,若是单通路,800MHz 的 RDRAM 带宽为 800MHz×16b÷8=1.6GB/s,若是两个通路,则可提升为 3.2GB/s,若是 4 个通路的话,将达到 6.4GB/s。而 DDR 133 的带宽为 133MHz×64b÷8=1.06GB/s,DDR 266 则为 2.1GB/s。

由于是全新的设计,需要用 RIMM 插槽与芯片组配合。RDRAM 总线是一条经过总线上所有设备(RDRAM 芯片)和模块的连接线路,每个模块在相对的两端有输入和输出引脚,时钟信号需依次流过每个 RIMM 槽,然后再通过每个 RIMM 槽返回。因此,任何不含 RDRAM 芯片的 RIMM 插槽必须填入一个连接模块即 Rambus 终结器以保证路径是完整的。

Rambus 虽然具有高带宽的优势,但只有在采用 Pentium 4 后的高性能微机上这种优势才能得到适当的发挥。目前,RDRAM 主要有 300MHz、356MHz 和 400MHz 3 种速率的产品,更高速率(533MHz)的产品也已经开发成功。由于 RDRAM 的双沿传输等同于速率的加倍,所以常把上述 3 种速率的 RDRAM 称为 PC-600、PC-700(实际上是 PC-711)和 PC-800 主存。

此外,RDRAM 还有一个特点,就是它的行地址与列地址的寻址总线是各自分离的独立总线,这就意味着行与列的选址几乎在同一时间内进行,从而进一步提高了工作效率;也正因为拥有这一优势,使得 RDRAM 不仅可以弥补它在寻址时间上比传统的 SDRAM 慢的缺点,而且在实际工作中所表现出来的性能更好。

但是,Rambus 最终并没有得到市场的认可,究其原因,就是因为 Rambus 内存高昂的售价以及“巨大”的发热量,加上 Rambus DRAM 必须安装两条才能够使用,这就大大提高了这种内存的使用门槛。最终,Rambus DRAM 没有经受住市场的考验,被价格更低的 DDR SDRAM 踩在了脚下。

5.5.8 多通道内存技术

多通道内存技术是解决 CPU 总线带宽与内存带宽矛盾的低价、高性能方案,其实质上是一种多通道内存控制和管理技术,与内存自身无关。目前双通道内存技术和三通道内存技术已在微机上广泛使用,四通道内存技术也已经出现,只不过目前价格还非常昂贵,大多在服务器和工作站中运用。

1. 双通道内存技术

双通道内存技术,就是在北桥芯片组里制作两个内存控制器,这两个内存控制器是可以相互独立工作的。在这两个内存通道上,CPU 可以分别寻址、读取数据,从而可以使内存的带宽增加一倍,理论上数据存取速度也相应增加一倍。

双通道 DDR 有两个 64 位内存控制器,双 64 位内存体系所提供的带宽等同于一个 128 位内存体系所提供的带宽,但是二者所达到效果却是不同的。因为双通道体系的两个内存控制器是独立的、具备互补性的智能内存控制器,两个内存控制器都能够在彼此间零等待时

间的情况下同时运行。例如,当控制器B准备进行下一次存取内存的时候,控制器A就在读写主内存,反之亦然。两个内存控制器的这种互补“天性”,可以让有效等待时间缩减50%,从而使内存的带宽翻了一番。

由于双通道内存技术将内存位宽扩大到128位,如使用双通道DDR 400内存,其内存带宽为400MHz×128b÷8=6.4GB/s;如使用双通道DDR2 800内存,其内存带宽将达到800MHz×128b÷8=12.8GB/s。

打开双通道模式必须要主板的北桥芯片或处理器支持。早期,双通道技术对内存条的要求十分严苛,两条规格(容量、时钟频率、延迟、颗粒、品牌、周期)必须相同。现在由于采用了不对称双通道(即采用了两个统一定址空间的存储器控制器),所以可以支持使用两条不同规格的内存条。

并非有支持双通道的主板上安装两条内存条就能运行工作,还需要正确的安装。只有当两组通道上都同时安装了内存条时,才能使内存工作在双通道模式下,否则只能工作在单通道模式。由于各家主板不同,因此必须要按照主板说明书以正确方式安装。

2. 三通道内存技术

随着Intel Core i7平台发布,三通道内存技术应运而生。Core i7处理器抛弃了前端总线而采用QPI总线,同时将内存控制从北桥中成功转移到CPU中,内存与处理器之间采用点对点连接设计,内存里的数据可由内存总线直接传送给处理器,使得内存读取延迟大幅减少。

三通道内存技术实际上是双通道内存技术的后续技术发展,三通道将内存总线位宽扩大到了64b×3=192b,同时采用DDR3 1333内存,因此其内存总线带宽达到了1333MHz×192b÷8=32GB/s,内存带宽得到巨大的提升。三通道内存的理论性能也能比同频率双通道内存提升50%以上。

对于支持三通道内存的主板,无论是4根内存插槽还是6根内存插槽的产品,要想实现三通道模式,只要将同色的三根内存插槽插上内存条即可,系统便会自动识别并进入三通道模式。但是,如果插上非3条或者非6条的内存,系统会自动进入单通道模式。

5.6 多体交叉存储技术

目前,主存的存取速度已成为计算机系统的瓶颈,除去通过寻找高速元件来提高访问速度外,也可以采用多个存储器并行工作,并且用交叉访问技术来提高存储器的访问速度。

5.6.1 并行访问存储器

常规的主存是单体单字存储器,只包含一个存储体。在高速的计算机中,普遍采用并行主存系统,即在一个存取周期内可以并行读出多个字,依靠整体信息吞吐率的提高,以解决CPU与主存之间的速度匹配问题。

多个并行工作的存储器共有一套地址寄存器和译码电路,按同一地址并行地访问各自的对应单元。例如,CPU送出地址A,则n个存储器中的所有A单元同时被选中。假设每个存储器的字长为w位,则同时访问$n\times w$位。也可以将这n个存储器看作一个大存储器,

一次访问 n 个字，故称为单体多字并行存储系统，如图 5-25 所示。

并行访问存储器按地址在一个存取周期内可读出 $n\times w$ 位的指令或数据，使主存带宽提高 n 倍。显然，采用这种方法的前提是：指令和数据在主存中必须是连续存放的，一旦遇到转移指令，或者操作数不能连续存放，这种方法的效果就不明显了。并行访问存储器的主要缺点是访问主存的冲突大。

5.6.2 交叉访问存储器

交叉访问存储器中有多个容量相同的存储模块（存储体），而且各存储模块具有各自独立的地址寄存器、读写电路和数据寄存器，这就是多体系统。各个存储体能并行工作，又能交叉工作。

多体交叉访问存储器如图 5-26 所示。存储器地址寄存器的低位部分经过译码选择不同的存储体，而高位部分则指向存储体内的存储字。现以由 4 个分体组成的多体交叉存储器为例，说明常用的编址方式。4 个分体 M_0、M_1、M_2、M_3 的编址序列如表 5-8 所示，称之为模 4 交叉编址序列。

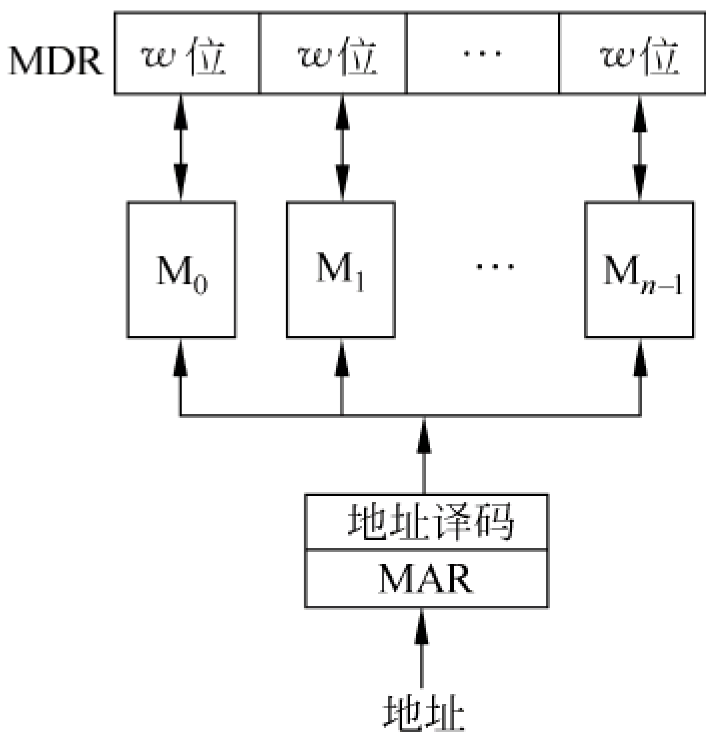


图 5-25　单体多字并行存储系统

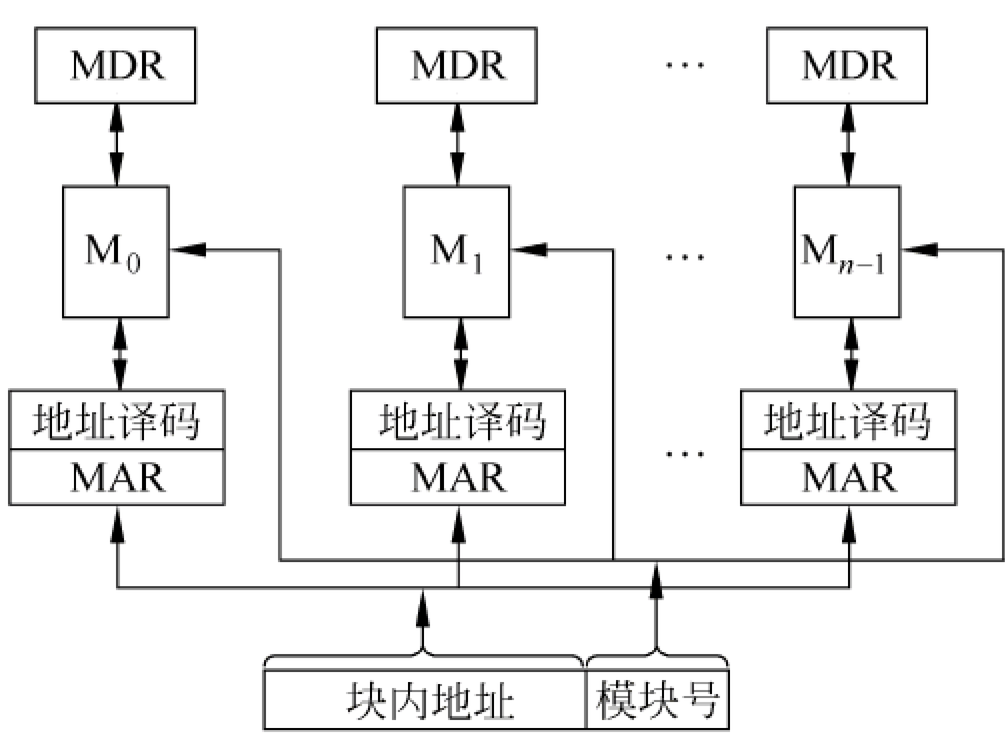


图 5-26　多体交叉访问存储器

表 5-8　模 4 交叉编址序列

模　块　号	地址编址序列	对应二进制地址的最低两位
M_0	0,4,8,12,…,4i+0,…	00
M_1	1,5,9,13,…,4i+1,…	01
M_2	2,6,10,14,…,4i+2,…	10
M_3	3,7,11,15,…,4i+3,…	11

在这种交叉存储器中，连续的地址分布在相邻的存储体中，而同一存储体内的地址都是不连续的。这种编址方式又称为横向编址。

多体交叉访问存储器采用分时启动的方法，可以在不改变每个模块存取周期的前提下，提高整个主存的速度。例如，有 4 个模块，在第一个存储周期的开始时刻启动模块 M_0，在 $\frac{T_m}{4}$、$\frac{T_m}{2}$、$\frac{3T_m}{4}$时刻分别启动模块 M_1、M_2、M_3，图 5-27 示意了模 4 交叉存取的时间关系。在 4 个分体完全并行的理想情况下，整个主存的有效周期缩小到原来模块存取周期的$\frac{1}{4}$，

数据传送的平均速度提高到原来的4倍。

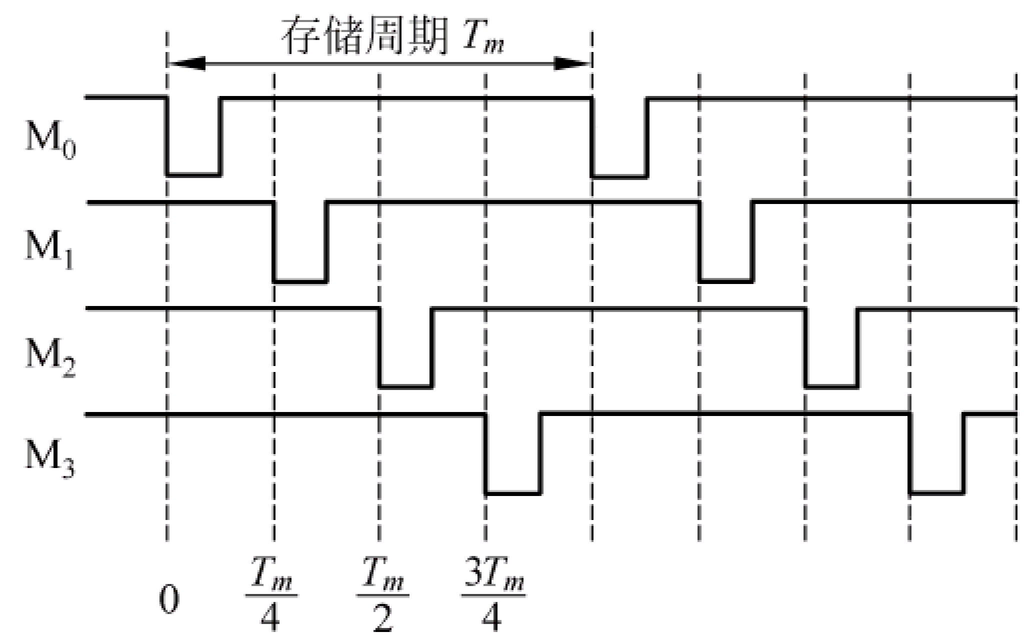


图 5-27 模4交叉存取分时工作

但是在实际应用中,当出现数据相关和程序转移时,将破坏并行性,不可能达到上述理想值。

注意:交叉访问存储器要求存储体的个数是2的整数幂,即必须是2、4、8、16…个,而且任一分体出现故障都将影响整个地址空间的所有区域。

5.7 高速缓冲存储器

主存速度的提高始终跟不上CPU的发展。据统计,CPU的速度平均每年提高60%,而组成主存的DRAM的速度平均每年只改进7%。由SRAM组成的高速缓冲存储器的运行速度则接近甚至等于CPU的速度。

5.7.1 高速缓存工作原理

1. 程序的局部性原理

程序的局部性有两个方面的含义:时间局部性和空间局部性。时间局部性是指如果一个存储单元被访问,则可能该单元会很快被再次访问。这是因为程序存在着循环。空间局部性是指如果一个存储单元被访问,则该单元邻近的单元也可能很快被访问。这是因为程序中大部分指令是顺序存储、顺序执行的,数据一般也是以向量、数组、树、表等形式簇聚地存储在一起的。

高速缓冲技术就是利用程序的局部性原理,把程序中正在使用的部分存放在一个高速的容量较小的Cache中,使CPU的访存操作大多数针对Cache进行,从而使程序的执行速度大大提高。

2. Cache的基本结构

图5-28给出了Cache的基本结构。Cache和主存都被分成若干个大小相等的块,每块由若干字节组成。由于Cache的容量远小于主存的容量,所以Cache中的块数要远少于主存中的块数,它保存的信息只是主存中最急需执行的若干块的副本。用主存地址的块号字段访问Cache标记,并将取出的标记和主存地址的标记字段相比较,若相

等，说明访问 Cache 有效，称 Cache 命中；若不相等，说明访问 Cache 无效，称 Cache 不命中或失效。

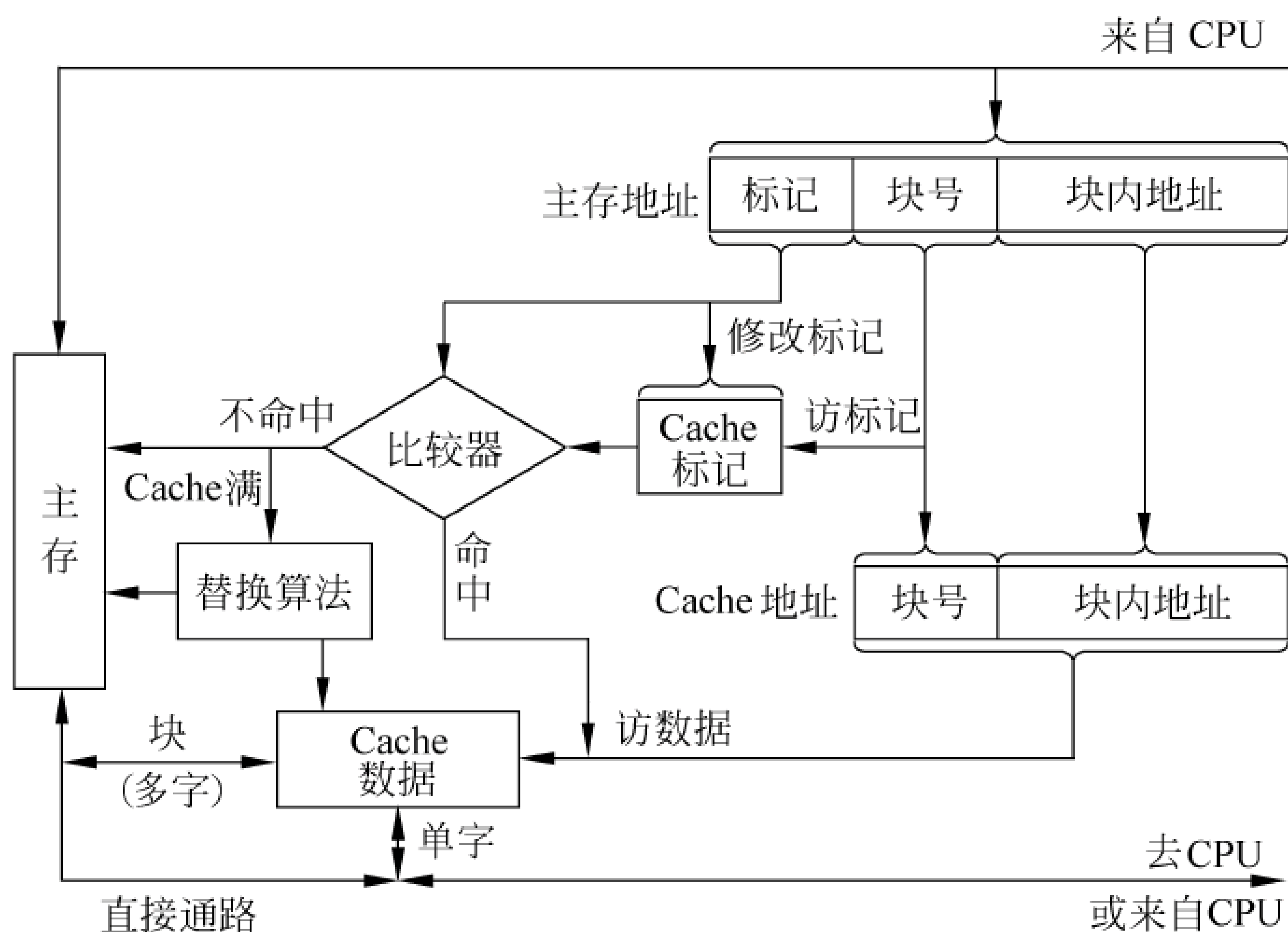


图 5-28　Cache 的基本结构

5.7.2　Cache 的读写操作

1. Cache 的读操作

当 CPU 发出读请求时，如果 Cache 命中，就直接对 Cache 进行读操作，与主存无关；如果 Cache 不命中，则仍需访问主存，并把该块信息一次从主存调入 Cache 内。若此时 Cache 已满，则必须根据某种替换算法，用这个块替换掉 Cache 中原来的某块信息。

2. Cache 的写操作

由于 Cache 中保存的只是主存的部分副本，这些副本与主存中的内容能否保持一致，是 Cache 能否可靠工作的一个关键问题。当 CPU 发出写请求时，如果 Cache 命中，有可能会遇到 Cache 与主存中的内容不一致的问题。例如，由于 CPU 写 Cache，把 Cache 某单元中的内容从 X 修改成了 X′，而主存对应单元中的内容仍然是 X，没有改变。所以如果 Cache 命中，需要进行一定的写处理，处理的方法有：写直达法和写回法，详见 5.7.5 节。

如果写 Cache 不命中，就直接把信息写入主存，并有两种处理方法：

① 不按写分配法，即只把所要写的信息写入主存。

② 按写分配法，即在把所要写的信息写入主存后还把这个块从主存中读入 Cache。

5.7.3　地址映像

在 Cache 中，地址映像是指把主存地址空间映像到 Cache 地址空间，也就是把存放在主存中的程序按照某种规则装入 Cache 中。地址映像的方法有 3 种：全相联映像、直接映像和组相联映像。

1. 全相联映像

全相联映像就是让主存中任何一个块均可以映像装入到 Cache 中任何一个块的位置上,如图 5-29(a)所示。全相联映像方式比较灵活,Cache 的块冲突概率最低、空间利用率最高,但是地址变换速度慢,而且成本高,实现起来比较困难。

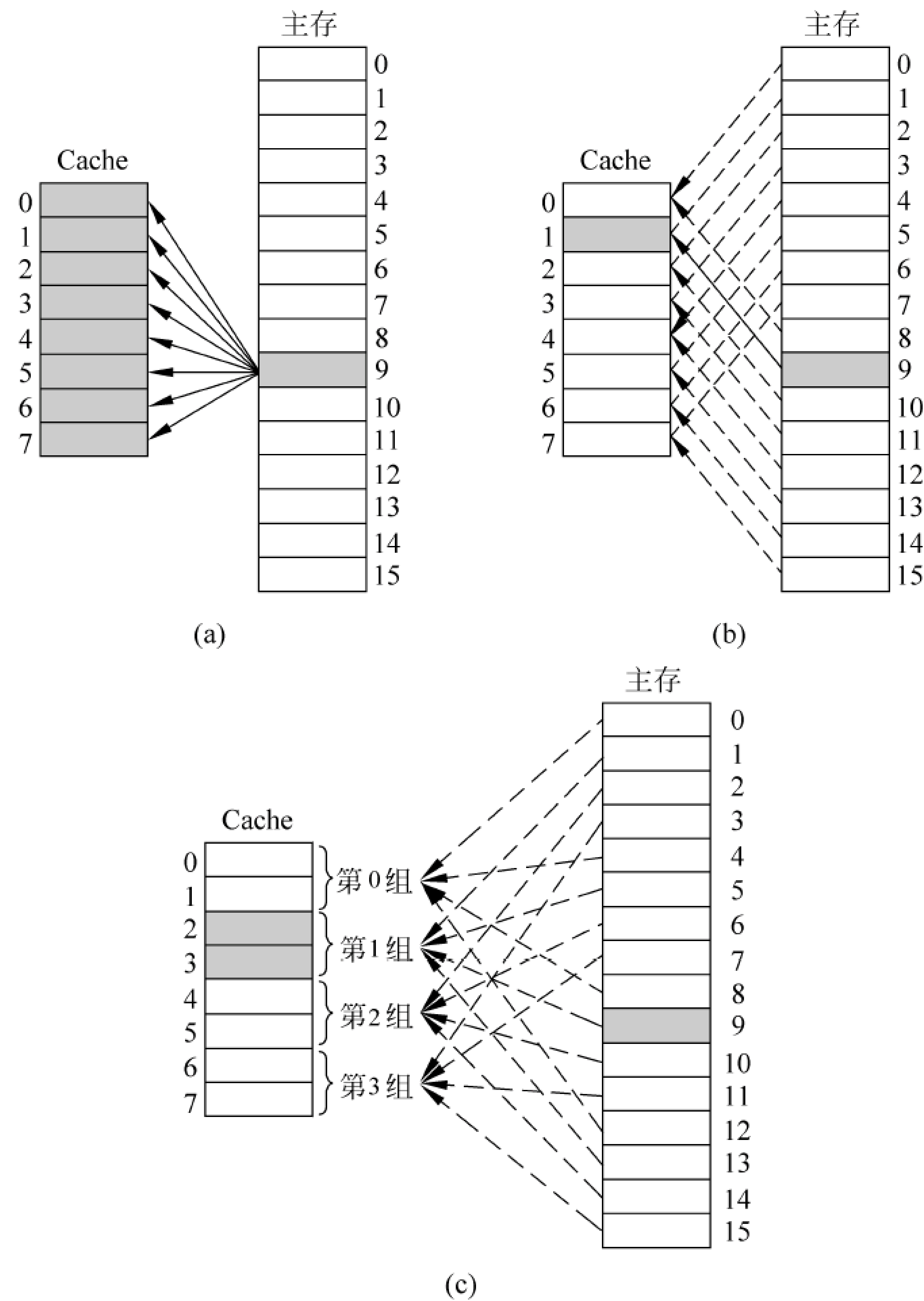


图 5-29 3 种映像规则

2. 直接映像

直接映像是指主存中的每一个块只能被放置到 Cache 中唯一的一个指定位置,若这个位置已有内容,则产生块冲突,原来的块将无条件地被替换出去。直接映像方式是最简单的地址映像方式,成本低,易实现,地址变换速度快,而且不涉及其他两种映像方式中的替换算法问题。但是,这种方式不够灵活,Cache 的块冲突概率最高,空间利用率最低。

直接映像规则如图 5-29(b)所示。例如,主存的第 0 块、第 8 块,只能映像到 Cache 的第 0 块;而主存的第 1 块、第 9 块,只能映像到 Cache 的第 1 块;等等。直接映像的关系可定

义为：

$$K = I \bmod 2^c$$

式中，K 为 Cache 的块号；I 为主存的块号；2^c 为 Cache 块数。

3. 组相联映像

组相联映像将 Cache 空间分成大小相同的组，让主存中的一块直接映像装入 Cache 中对应组的任何一块位置上，即组间采取直接映像，而组内采取全相联映像。

当组相联映像中组内的块容量为 1 时，就转化为直接映像；当组内的块容量为 Cache 的容量时，就转化为全相联映像。因此，组相联映像实际上是全相联映像和直接映像的折中方案，其优点和缺点介于全相联和直接映像方式的优缺点之间。

组相联映像规则如图 5-29(c)所示。Cache 分为 4 组，每组 2 块。主存的第 9 块将可以映像到 Cache 第 1 组的位置上。组相联映像的关系可以定义为：

$$J = I \bmod Q$$

式中，J 为 Cache 的组号；I 为主存的块号；Q 为 Cache 的组数。

5.7.4 替换算法

在采用全相联映像和组相联映像方式从主存向 Cache 传送一个新块，而 Cache 中的空间已被占满时，就需要把原来存储的一块替换掉。常用的替换算法有下述 3 种。

1. 随机算法

最简单的替换算法是随机方法。随机法完全不管 Cache 块过去、现在及将来的使用情况，简单地根据一个随机数，选择一块替换掉。

2. 先进先出(FIFO)算法

FIFO 算法的思想是：按调入 Cache 的先后决定淘汰的顺序，即在需要更新时，将最先进入 Cache 的块作为被替换的块。这种方法要求为每块做一记录，记下它们进入 Cache 的先后次序。这种方法容易实现，而且系统开销小。其缺点是可能会把一些需要经常使用的程序块(如循环程序)也作为最早进入 Cache 的块替换掉。

3. 近期最少使用(LRU)算法

LRU 算法是把 CPU 近期最少使用的块作为被替换的块。这种替换方法需要随时记录 Cache 中各块的使用情况，以便确定哪个块是近期最少使用的块。LRU 算法相对合理，但实现起来比较复杂，系统开销较大。通常需要对每一块设置一个称为“年龄计数器”的硬件或软件计数器，用于记录其被使用的情况。

5.7.5 更新策略

为了解决 Cache 与主存中内容不一致问题，首先要选择合适的 Cache 更新策略。Cache 有两种更新策略：写直达法和写回法。

写直达法是指 CPU 在执行写操作时，必须把数据同时写入 Cache 和主存。当某一块需

要替换时,也不必把这一块写回到主存中,新调入的块可以立即把这一块覆盖。这种方法实现简单,而且能随时保持主存数据的正确性,但可能增加多次不必要的主存写入,会降低存取速度。

写回法是指CPU在执行写操作时,被写数据只写入Cache,不写入主存。仅当需要替换时,才把已经修改过的Cache块写回到主存。在采用这种更新策略的Cache块表中,一般有一个标志位(称为"脏"位),当一块中的任何一个单元被修改时,标志位被置"1"。在需要替换掉这一块时,如果标志位为"1"(表示该块数据已经"脏"了),则必须先把这一块写回到主存中之后,才能再调入新的块;如果标志位为"0"(表示该块数据还是干净的),则这一块不必写回主存,只要用新调入的块覆盖掉这一块即可。这种方法操作速度快,但因主存中的字块未及时修改而有可能出错。

5.7.6 微机中Cache技术的实现

Cache刚出现时,典型系统中只有一个Cache,而近年来的微机系统中普遍采用了多个Cache。多个Cache的含义有两方面:一是增加Cache的级数;二是将统一的Cache变成分开的Cache。下面以PC为例介绍Cache技术的实现。

1. 单一缓存和多级缓存

所谓单一缓存,顾名思义是在CPU和主存之间只设一个Cache。80386以前的CPU只有外部的Cache。

随着集成电路技术的发展,Cache被直接与CPU制作在同一个芯片内,称为一级Cache(L1 Cache),又称片内Cache。L1 Cache总是以CPU的核心速度运行,是所有系统中最快的高速缓存。CPU直接访问L1 Cache,不必占用外部总线,而且L1 Cache与CPU之间的数据通路很短,所以大大提高了存取速度,但L1 Cache容量较小,一般仅为几十KB。而此时安装在主板上的Cache则称为L2 Cache(二级缓存),L2 Cache以主板速度运行,可以有较大的容量,从256KB到2MB不等。从Pentium Pro和Pentium Ⅱ微处理器开始,将L2 Cache和CPU封装在一起,最初速度为CPU核心速度的一半、五分之二或三分之一。后来,大多数CPU芯片内都同时集成了L1和L2 Cache,并都能以全核心速度运行,被称为片内两级Cache设计。之后,为读取L2 Cache后未命中的数据又设计了L3 Cache,在拥有三级缓存的CPU中,只有约5%的数据需要从内存中调用,这进一步提高了CPU的效率,一开始L3 Cache是外置的,到2012年以后都是内置的,即将L3 Cache也封装在处理器盒内。目前,主流CPU的L2 Cache一般为512KB到6MB不等,L3 Cache一般为3MB到24MB不等。

多级Cache的技术难度和制造成本是相对递减的,所以其容量也是相对递增的。CPU访存时首先查找L1 Cache,如果L1 Cache不命中,则访问L2 Cache……直到所有级别的Cache都不命中时,才访问主存。假设,在拥有两级Cache的CPU中,读取L1 Cache的命中率为80%,读取L2的命中率也在80%(从L2读到有用的数据占总数据的16%),剩下的数据就不得不从主存中调用了,但这只占总数据的4%。

2. 统一缓存和分开缓存

统一缓存是指指令和数据都存放在同一个 Cache 中。而分开缓存是指指令和数据分别存放在两个 Cache 中，一个称为指令 Cache，另一个称为数据 Cache。

80486 及其以前的微处理器都只有统一缓存，由于 Pentium 开始采用流水线控制技术，所以必须将指令和数据的 Cache 分开，以满足指令预取和指令执行并行的需要，否则将可能出现取指和执行过程对统一缓存的争用。目前，绝大多数 CPU 中的 L1 Cache 都将指令 Cache 和数据 Cache 分开，成为典型的哈佛结构。一般情况下，L1 数据 Cache 和 L1 指令 Cache 的容量相同，二者分别用来存放数据和指令，并对执行这些数据的指令进行即时解码。例如，AMD 的 Athlon XP 就具有 64KB 的 L1 数据 Cache 和 64KB 的 L1 指令 Cache，其 L1 Cache 就以 64KB+64KB 来表示。但 Pentium 4 的 L1 Cache 有点特殊，它使用新增加的一种一级追踪缓存(Execution Trace Cache，ETC)来替代指令 Cache，容量为 12KμOps(表示能存储 12K 条微指令，有关微指令的概念将在第 6 章中详细讨论)，而其数据 Cache 只有 8KB，故其一级缓存就以 8KB+12KμOps 表示。

一级追踪缓存与一级指令缓存的运行机制是不相同的，一级追踪缓存能将微指令储存在一级追踪缓存之内，所以有效地增加了高工作频率下对指令的解码能力。

注意：在这里 12KμOps 绝对不等于 12KB，二者单位不同，运行机制不同，简单相加得出 L1 Cache 容量的看法是完全错误的。

除去 L1 Cache 以外，L2 Cache 和 L3 Cache 都只存储数据。

5.8 虚拟存储器

虚拟存储器由主存储器和联机工作的辅助存储器(通常为磁盘存储器)共同组成，这两个存储器在硬件和系统软件的共同管理下工作，对于应用程序员，可以把它们看作一个单一的存储器。

5.8.1 虚拟存储器的基本概念

虚拟存储器将主存或辅存的地址空间统一编址，形成一个庞大的存储空间。在这个大空间里，用户可以自由编程，完全不必考虑程序在主存是否装得下以及这些程序将来在主存中的实际存放位置。

用户编程的地址称为虚地址或逻辑地址，实际的主存单元地址称为实地址或物理地址。显然，虚地址要比实地址大得多。

在实际的物理存储层次上，所编程序和数据在操作系统管理下，先送入磁盘，然后操作系统将当前运行所需要的部分调入主存，供 CPU 使用，其余暂不运行部分留在磁盘中。

程序运行时，CPU 以虚地址来访问主存，由辅助硬件找出虚地址和实地址之间的对应关系，并判断这个虚地址指示的存储单元内容是否已装入主存。如果已在主存中，则通过地址变换，CPU 可直接访问主存的实际单元；如果不在主存中，则把包含这个字的一页或一个程序段调入主存后再由 CPU 访问。如果主存已满，则由替换算法从主存中将暂不运行的一页或一段调回辅存，再从辅存调入新的一页或一段到主存。从原理的角度看，虚拟存储器

和 Cache-主存有不少相同之处。事实上,前面提到的各种方法是先应用于虚拟存储器中,后来才发展到 Cache-主存层次中的。

5.8.2 页式虚拟存储器

以页为基本单位的虚拟存储器称为页式虚拟存储器。各类计算机页面大小不等,一般为 512B 到几 KB。主存空间和虚存空间都划分成若干个大小相等的页。主存即实存的页称为实页,虚存的页称为虚页。

程序虚地址分为两个字段:高位字段为虚页号,低位字段为页内地址。虚地址到实地址之间的变换是由页表来实现的。页表是一张存放在主存中的虚页号和实页号的对照表,记录着程序的虚页调入主存时被安排在主存中的位置。若计算机采用多道程序工作方式,则可为每个用户作业建立一个页表,硬件中设置一个页表基址寄存器,存放当前所运行程序的页表的起始地址。

页表中的每一行记录了与某个虚页对应的若干信息,包括虚页号、装入位和实页号等。页表基址寄存器和虚页号拼接成页表索引地址。根据这个索引地址可读到一个页表信息字,然后检测页表信息字中装入位的状态。若装入位为“1”,则表示该页面已在主存中,将对应的实页号与虚地址中的页内地址相拼接就得到了完整的实地址;若装入位为“0”,则表示该页面不在主存中,于是要启动 I/O 系统,把该页从辅存中调入主存后再供 CPU 使用。图 5-30 给出了页式虚拟存储器的虚-实地址的变换过程。

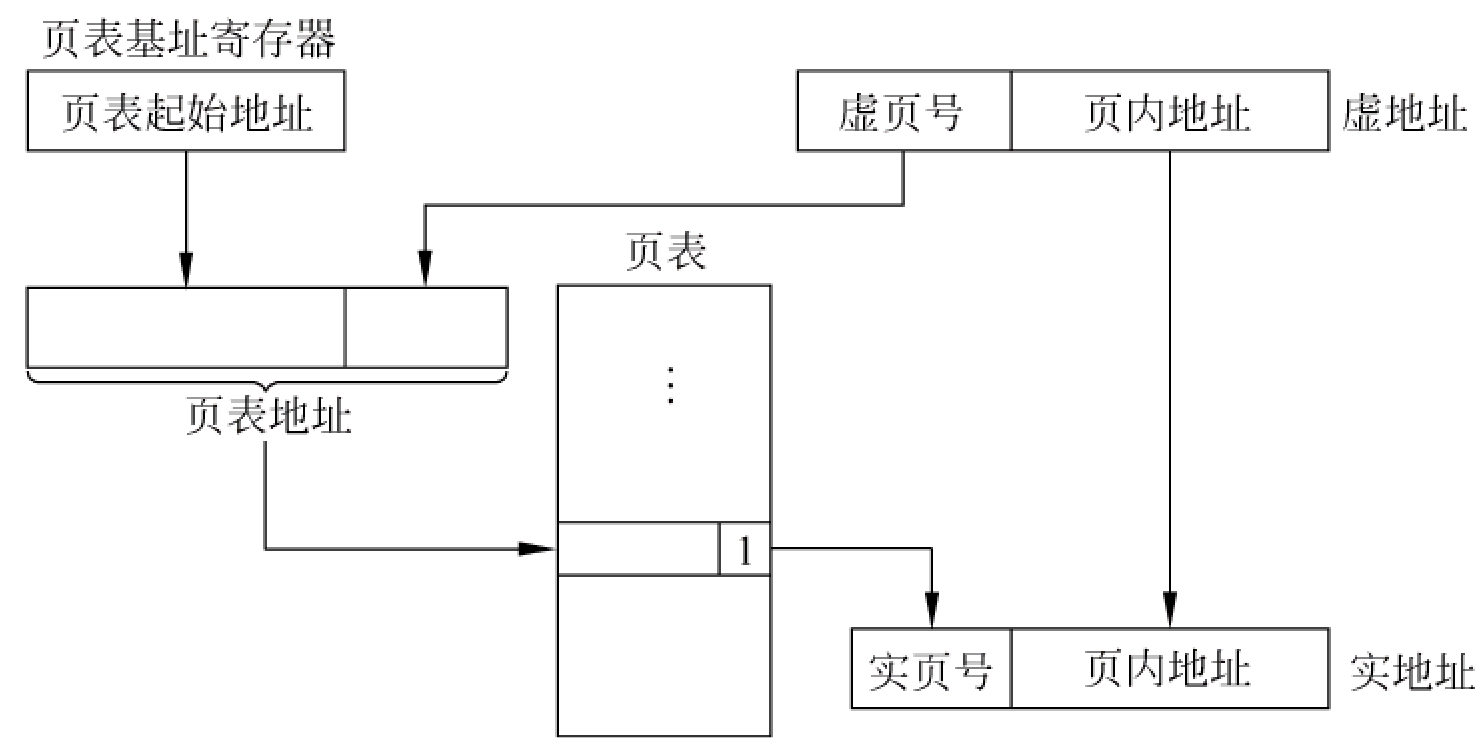


图 5-30 页式虚拟存储器的虚-实地址变换

从上述地址转换过程可知,CPU 访存时首先要查页表,为此需要访问一次主存,若不命中,还要进行页面替换和页表修改,则访问主存的次数就更多了。

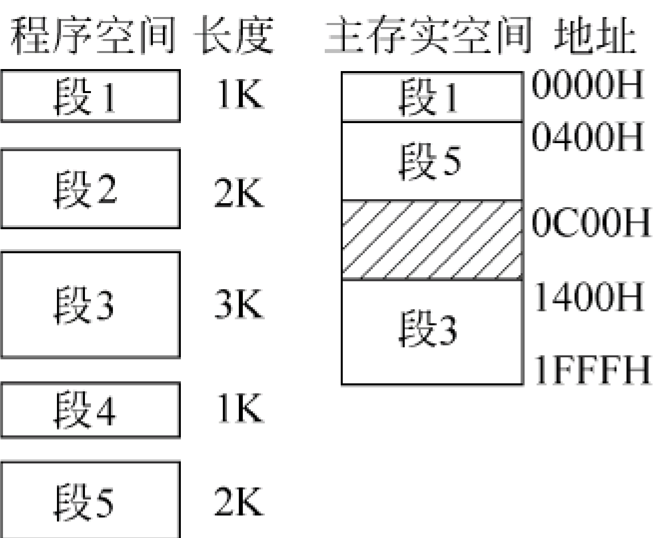
页式虚拟存储器的每页长度是固定的,页表的建立很方便,新页的调入也容易实现。但是,由于程序不可能正好是页面的整倍数,最后一页的零头将无法利用而造成浪费。同时,页不是逻辑上独立的实体,使得程序的处理、保护和共享都比较麻烦。

5.8.3 段式虚拟存储器

段式虚拟存储器中的段是按照程序的逻辑结构划分的,各个段的长度因程序而异。为了把程序虚地址变换成主存实地址,需要一个段表。段表中每一行记录了某个段对应的若干信息,包括段号、装入位、段起点和段长等。段表一般驻留在主存中。若装入位为“1”,则

表示该段已调入主存；若装入位为“0”，则表示该段不在主存中。由于段的大小可变，所以在段表中要给出各段的起始地址与段的长度。段表实际上是程序的逻辑结构段与其在主存中所存放的位置之间的关系对照表，如图 5-31 所示。

编程使用的虚地址包含两部分：段号和段内地址。段式虚拟存储器的虚-实地址的变换过程如图 5-32 所示。CPU 根据虚地址访存时，首先将段号与段表的起始地址相拼接，形成访问段表对应行的地址，然后根据段表内装入位判断该段是否已调入主存。若已调入主存，从段表读出该段在主存中的起始地址，与段内地址（偏移量）相加，得到对应的主存实地址。



段表

段号	段起点	装入位	段长
1	0000H	1	1K
2		0	
3	1400H	1	3K
4		0	
5	0400H	1	2K

图 5-31　程序在主存中的分配及其段表

由于段的分界与程序的自然分界相对应，所以具有逻辑独立性，易于程序的编译、管理、修改和保护，也便于多道程序共享。但是，因为段的长度参差不齐，起点和终点不定，给主存空间分配带来了麻烦；容易在段间留下不能利用的零头，造成浪费。

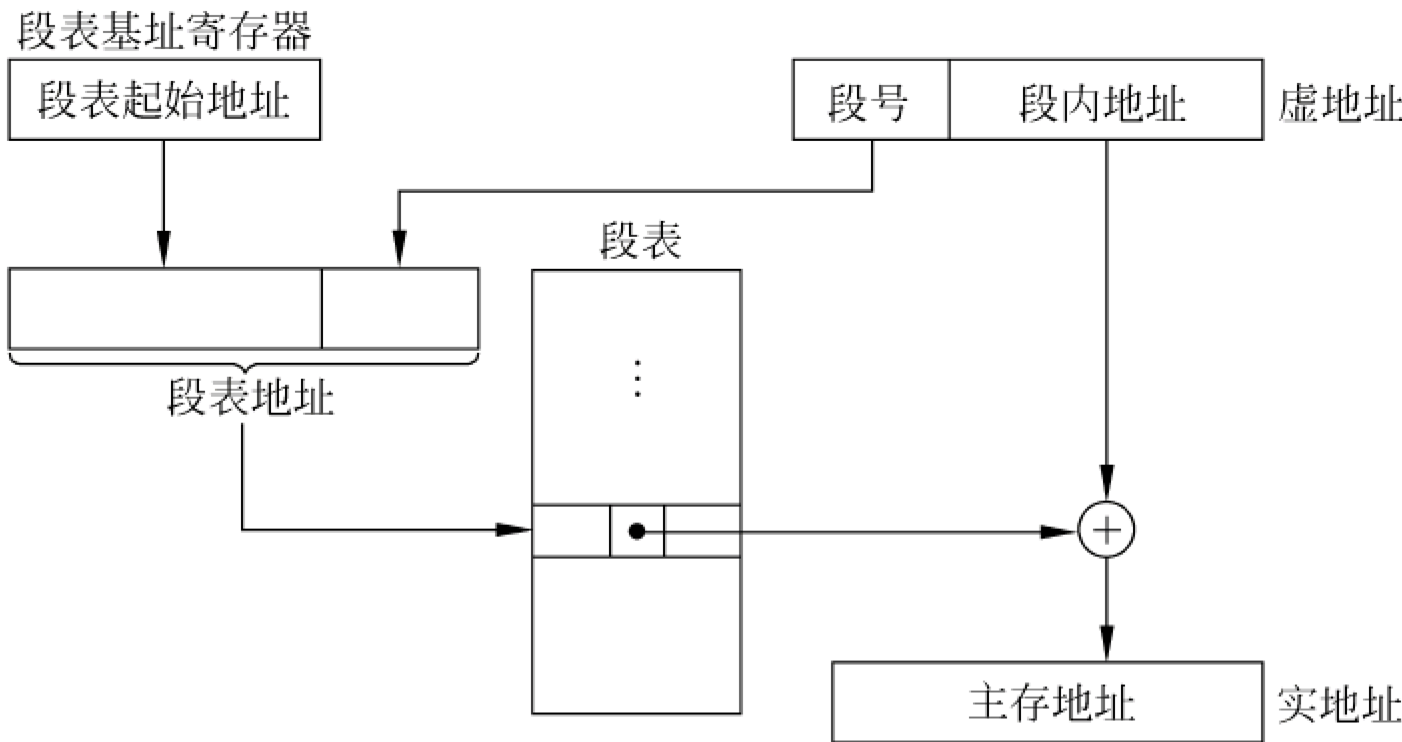


图 5-32　段式虚拟存储器的虚-实地址变换

5.8.4　段页式虚拟存储器

在段式、页式存储器的基础上，还有一种段页式虚拟存储器。将程序按其逻辑结构分段，每段再划分为若干大小相等的页；主存空间也划分为若干同样大小的页。虚存和实存之间以页为基本传送单位，每个程序对应一个段表，每段对应一个页表。CPU 访问时，虚地址包含段号、段内页号、页内地址 3 部分。首先将段表起始地址与段号合成，得到段表地址；然后从段表中取出该段的页表起始地址，与段内页号合成，得到页表地址；最后从页表中取出实页号，与页内地址拼接形成主存实地址。段页式存储器综合了前两种结构的优点，但要经过两级查表才能完成地址转换，费时要多些。

段页式虚拟存储器将存储空间按逻辑模块分成段，每段又分成若干个页，访存通过一个段表和若干个页表进行。段的长度必须是页长的整数倍，段的起点必须是某一页的起点。

5.8.5 快表与慢表

在虚拟存储器中,如果不采取有效的措施,访问主存的速度将要降低几倍,这是因为在页式或段式虚拟存储器中,必须先查页表或段表;在段页式虚拟存储器中,既要查段表也要查页表。

要想使访问虚存的速度接近于访问主存的速度,必须加快查表的速度。由于程序在执行过程中具有局部性的特点,因此对页表的访问并不完全是随机的。在一段时间内,对页表的访问只是局限在少数几个存储器字内。为了将访问页表的时间降低到最低限度,许多计算机将页表分为快表和慢表两种。将当前最常用的页表信息存放在一个小容量的高速存储器中,称为“快表”(TLB),当快表中查不到时,再从存放在主存中的页表中查找实页号。与快表相对应,存放在主存中的页表称为“慢表”。快表只是慢表的一个副本,而且只存放了慢表中很少的一部分。

实际上,快表与慢表也构成了一个由两级存储器组成的存储系统,其访问速度接近于快表的速度,存储容量是慢表的容量。快表容量很小(几十个字),速度高,采用相联方式,按内容访问。

习　题

5-1　如何区别存储器和寄存器?两者是一回事的说法对吗?

5-2　存储器的主要功能是什么?为什么要把存储系统分成若干个不同层次?主要有哪些层次?

5-3　在一个字节编址的计算机中,假定int型变量i的地址为0200H,i的机器数为01234567H,请用表格的方式分别列出大端方案和小端方案情况下各个字节对应的主存地址。

5-4　某机存储字长64位,主存储器按字节编址,现有4种不同长度的数据:字节、半字(16位)、单字(32位)、双字(64位),请采用一种既节省存储空间,又能保证任一个数据都在单个存取周期中完成读写的方法,将不同长度的数据存入主存(采用大端方案)。

(1) 写出不同长度数据存放在主存中地址的限定要求(即第一个字节的地址)。

(2) 画出将字节、双字、半字、单字、字节5个数据依次存放在主存中的示意图(不能改变顺序)。

5-5　动态RAM为什么要刷新?一般有几种刷新方式?各有什么优缺点?

5-6　一般存储芯片都设有片选端$\overline{CS}$,它有什么用途?

5-7　DRAM芯片和SRAM芯片通常有何不同?

5-8　有哪几种只读存储器?它们各自有何特点?

5-9　说明存取周期和存取时间的区别。

5-10　一个1K×8的存储芯片需要多少根地址线、数据输入线和输出线?

5-11　某计算机字长为32位,其存储容量是64KB,按字编址的寻址范围是多少?若主存以字节编址,试画出主存字地址和字节地址的分配情况。

5-12　一个容量为16K×32的存储器,其地址线和数据线的总和是多少?当选用下列不同规格的存储芯片时,各需要多少片?

1K×4,2K×8,4K×4,16K×1,4K×8,8K×8

5-13　现有1024×1的存储芯片,若用它组成容量为16K×8的存储器。试求:

(1) 实现该存储器所需的芯片数量。

(2) 若将这些芯片分装在若干块板上,每块板的容量为4K×8,该存储器所需的地址线总位数是多少?

其中几位用于选板？几位用于选片？几位用作片内地址？

5-14　已知某计算机字长8位，现采用半导体存储器作主存，其地址线为16位，若使用1K×4的SRAM芯片组成该机所允许的最大主存空间，并采用存储模板结构形式。

(1) 若每块模板容量为4K×8，共需多少块存储模板？

(2) 画出一个模板内各芯片的连接逻辑图。

5-15　某半导体存储器容量16K×8，可选SRAM芯片的容量为4K×4；地址总线A_{15}～A_0(A_0为最低位)，双向数据总线D_7～D_0(D_0为最低位)，由$R/\overline{W}$线控制读写。设计并画出该存储器的逻辑图，并注明地址分配、片选逻辑及片选信号的极性。

5-16　现有如下存储芯片：2K×1的ROM、4K×1的RAM、8K×1的ROM。若用它们组成容量为16KB的存储器，前4KB为ROM，后12KB为RAM，CPU的地址总线16位。

(1) 各种存储芯片分别用多少片？

(2) 正确选用译码器及门电路，并画出相应的逻辑结构图。

(3) 指出有无地址重叠现象。

5-17　用容量为16K×1的DRAM芯片构成64KB的存储器。

(1) 画出该存储器的结构框图。

(2) 设存储器的读写周期均为0.5μs，CPU在1μs内至少要访存一次，试问采用哪种刷新方式比较合理？相邻两行之间的刷新间隔是多少？对全部存储单元刷新一遍所需的实际刷新时间是多少？

5-18　有一个8位机，采用单总线结构，地址总线16位(A_{15}～A_0)，数据总线8位(D_7～D_0)，控制总线中与主存有关的信号有$\overline{MREQ}$(低电平有效允许访存)和$R/\overline{W}$(高电平为读命令，低电平为写命令)。

主存地址分配如下：从0～8191为系统程序区，由ROM芯片组成；从8192～32767为用户程序区；最后(最大地址)2K地址空间为系统程序工作区(上述地址均用十进制表示，按字节编址)。

现有如下存储芯片：8K×8的ROM，16K×1、2K×8、4K×8、8K×8的SRAM。从上述规格中选用芯片设计该机主存储器，画出主存的连接框图，并注意画出片选逻辑及与CPU的连接。

5-19　某半导体存储器容量为15KB，其中固化区为8KB，可选EPROM芯片为4K×8；可随机读写区为7KB，可选SRAM芯片有：4K×4、2K×4、1K×4。地址总线A_{15}～A_0(A_0为最低位)，双向数据总线D_7～D_0(D_0为最低位)，$R/\overline{W}$为控制读写，$\overline{MREQ}$为低电平时允许存储器工作信号。设计并画出该存储器逻辑图，注明地址分配、片选逻辑、片选信号极性等。

5-20　某计算机地址总线16位A_{15}～A_0(A_0为最低位)，访存空间64KB。外围设备与主存统一编址，I/O空间占用FC00～FFFFH。现用2164芯片(64K×1)构成主存储器，设计并画出该存储器逻辑图，并画出芯片地址线、数据线与总线的连接逻辑以及行选信号与列选信号的逻辑式，使访问I/O时不访问主存。动态刷新逻辑可以暂不考虑。

5-21　已知有16K×1的DRAM芯片，其引脚功能如下：地址输入A_6～A_0，行地址选择$\overline{RAS}$，列地址选择$\overline{CAS}$，数据输入端D_{in}，数据输出端D_{out}，控制端$\overline{WE}$。请用给定芯片构成256KB的存储器，采用奇偶校验，试问：需要芯片的总数是多少？并完成下列问题。

(1) 正确画出存储器的连接框图。

(2) 写出各芯片$\overline{RAS}$和$\overline{CAS}$形成条件。

(3) 若芯片内部采用128×128矩阵排列，求异步刷新时该存储器的刷新周期。

5-22　并行存储器有哪几种编址方式？简述低位交叉编址存储器的工作原理。

5-23　什么是高速缓冲存储器？它与主存是什么关系？其基本工作过程如何？

5-24　Cache做在CPU芯片内有什么好处？将指令Cache和数据Cache分开又有什么好处？

5-25　设某计算机主存容量为4MB，Cache容量为16KB，每块包含8个字，每字32位，设计一个4路组相联映像(即Cache每组内共有4个块)的Cache组织，要求：

(1) 画出主存地址字段中各段的位数。

(2) 设Cache的初态为空,CPU依次从主存第0,1,2,…,99号单元读出100个字(主存一次读出一个字),并重复按此次序读8次,问命中率是多少?

(3) 若Cache的速度是主存的6倍,试问有Cache和无Cache相比,速度提高多少倍?

5-26　什么叫虚拟存储器?采用虚拟存储技术能解决什么问题?

5-27　已知采用页式虚拟存储器,某程序中一条指令的虚地址是:000001111111100000。该程序的页表起始地址是0011,页面大小为1K,页表中有关单元最末4位(实页号)如表5-9所示。

表5-9　页表

虚　页　号	装　入　位	实　页　号
007H	1	0001
⋮	⋮	⋮
300H	1	0011
⋮	⋮	⋮
307H	1	1100

指出指令地址(虚地址)变换后的主存实地址。

第6章 中央处理器

中央处理器(CPU)是整个计算机的核心,它包括运算器和控制器。本章着重讨论CPU的功能和组成,控制器的工作原理和实现方法,微程序控制原理,基本控制单元的设计,以及先进的CPU系统设计技术。

6.1 中央处理器的功能和组成

CPU对整个计算机系统的运行是极其重要的,这里将从CPU的功能、内部结构和主要技术参数入手,为后面详细讨论程序的执行过程打下基础。

6.1.1 CPU的功能

若用计算机来解决某个问题,首先要为这个问题编制解题程序,而程序又是指令的有序集合。按"存储程序"的概念,只要把程序装入主存储器后,即可由计算机自动地完成取指令和执行指令的任务。在程序运行过程中,在计算机的各部件之间流动的指令和数据形成了指令流和数据流。

注意:这里的指令流和数据流都是程序运行的动态概念,它不同于程序中静态的指令序列,也不同于存储器中数据的静态分配序列。指令流指的是CPU执行的指令序列,数据流指的是根据指令操作要求依次存取数据的序列。从程序运行的角度来看,CPU的基本功能就是对指令流和数据流在时间与空间上实施正确的控制。

对于冯·诺依曼结构的计算机而言,数据流是根据指令流的操作而形成的,也就是说数据流是由指令流来驱动的。

6.1.2 CPU中的主要寄存器

CPU中的寄存器是用来暂时保存运算和控制过程中的中间结果、最终结果以及控制、状态信息的,它可分为通用寄存器和专用寄存器两大类。

1. 通用寄存器

通用寄存器可用来存放原始数据和运算结果,有的还可以作为变址寄存器、计数器、地址指针等。现代计算机中为了减少访问存储器的次数,提高运算速度,往往在CPU中设置大量的通用寄存器,少则几个,多则几十个,甚至上百个。通用寄存器可以由程序编址访问。

累加寄存器 Acc 也是一个通用寄存器,它用来暂时存放 ALU 运算的结果信息。例如,在执行一个加法运算前,先将一个操作数暂时存放在 Acc 中,再从主存中取出另一操作数,然后同 Acc 的内容相加,所得的结果送回 Acc 中。运算器中至少要有一个累加寄存器。

2. 专用寄存器

专用寄存器是专门用来完成某一种特殊功能的寄存器。CPU 中至少要有 5 个专用的寄存器:程序计数器(PC)、指令寄存器(IR)、存储器数据寄存器(MDR)、存储器地址寄存器(MAR)和程序状态字寄存器(PSWR)。

(1) 程序计数器

程序计数器又称为指令计数器,用来存放正在执行的指令地址或者接着要执行的下一条指令地址。

对于顺序执行的情况,程序计数器的内容应不断地增量(加“1”),以控制指令的顺序执行。这种加“1”的功能,有些机器是程序计数器本身具有的,也有些机器是借助运算器来实现的。

在遇到需要改变程序执行顺序的情况时,将转移的目标地址送往程序计数器,即可实现程序的转移。

(2) 指令寄存器

指令寄存器用来存放从存储器中取出的指令。当指令从主存取出存于指令寄存器之后,在执行指令的过程中,指令寄存器的内容不允许发生变化,以保证实现指令的全部功能。

(3) 存储器数据寄存器

存储器数据寄存器用来暂时存放由主存储器读出的一条指令或一个数据字;反之,当向主存写入一条指令或一个数据字时,也暂时将它们存放在存储器数据寄存器中。

(4) 存储器地址寄存器

存储器地址寄存器用来保存当前 CPU 所访问的主存单元的地址。由于主存和 CPU 之间存在着操作速度上的差别,所以必须使用地址寄存器来保持地址信息,直到主存的读写操作完成为止。

当 CPU 和主存进行信息交换,无论是 CPU 向主存存取数据时,还是 CPU 从主存中读出指令时,都要使用存储器地址寄存器和数据寄存器。

(5) 程序状态字寄存器

程序状态字寄存器又称状态标志寄存器,用来存放程序状态字(PSW)。程序状态字的各位表征程序和机器运行的状态,是参与控制程序执行的重要依据之一。它主要包括两部分内容:一是状态标志,如进位标志(C)、结果为零标志(Z)等,大多数指令的执行将会影响到这些标志位;二是控制标志,如中断标志、陷阱标志等。程序状态字寄存器的位数往往等于机器字长,各类机器的程序状态字寄存器的位数和设置位置不尽相同。例如,8086 微处理器的程序状态字寄存器有 16 位,如图 6-1 所示,一共包括 9 个标志位,其中 6 个为状态标志,3 个为控制标志。

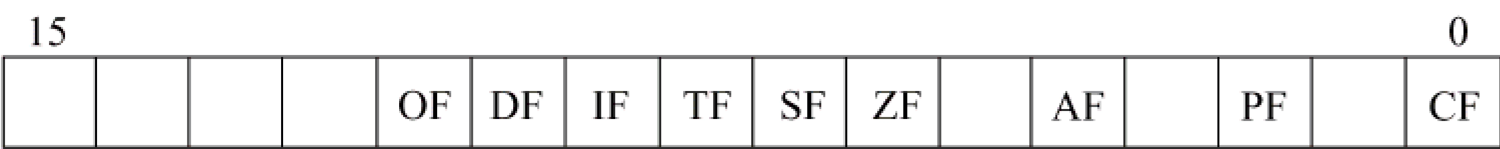


图 6-1 8086 微处理器的状态字寄存器

6个状态标志为：

- 进位标志位(CF)；
- 辅助进位标志位(AF)；
- 溢出标志位(OF)；
- 零标志位(ZF)；
- 符号标志位(SF)；
- 校验标志位(PF)。

3个控制标志为：

- 方向标志(DF)，表示串操作指令中字符串操作的方向；
- 中断允许标志位(IF)，表示CPU是否能够响应外部的可屏蔽中断请求；
- 陷阱标志位(TF)，为了方便程序的调试，使处理器的执行进入单步方式而设置的控制标志位。

6.1.3 CPU的组成

CPU由运算器和控制器两大部分组成，图6-2给出了CPU的模型。

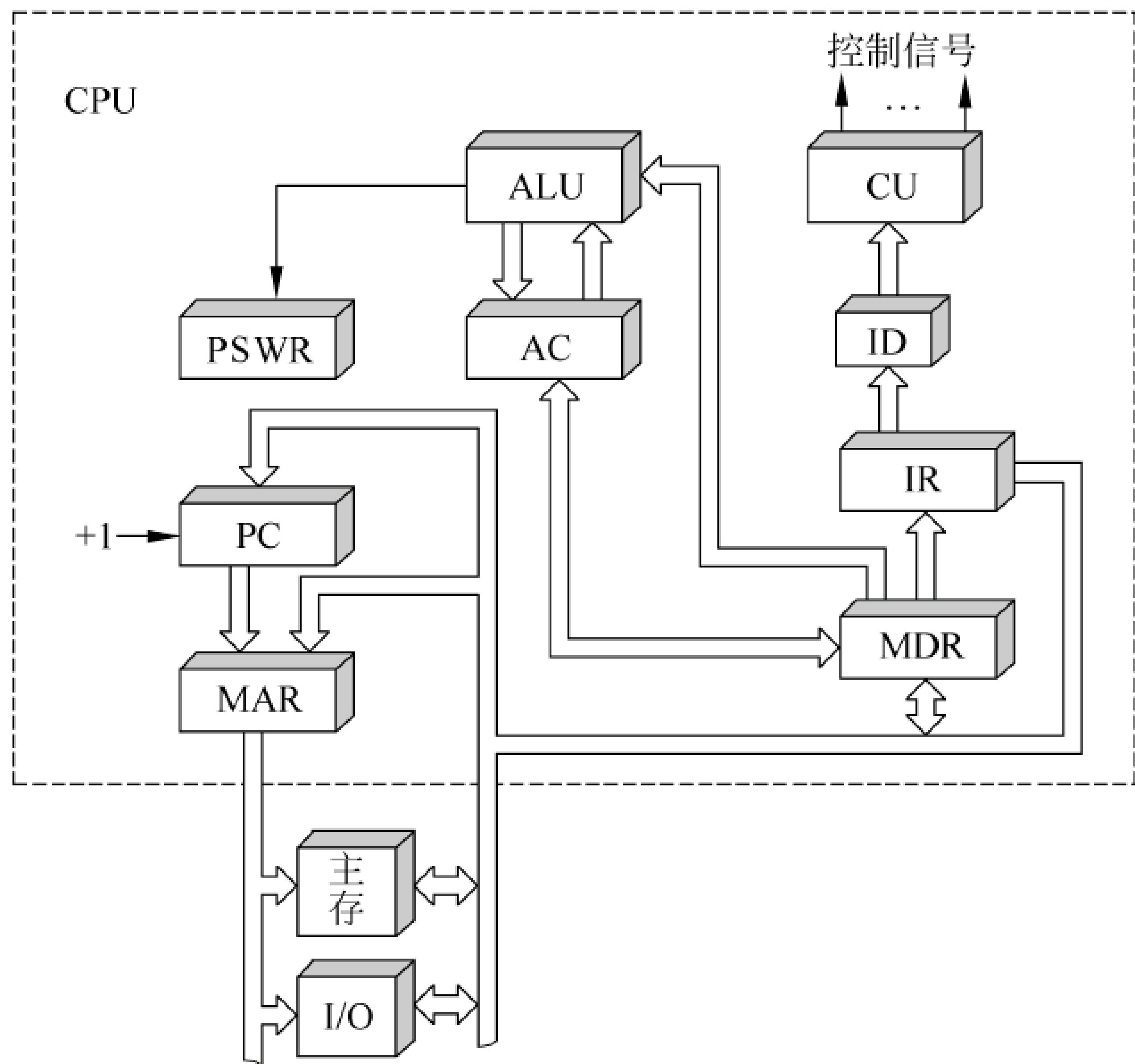


图6-2　CPU模型

在图6-2中，ID表示指令译码器，CU表示控制单元，其作用将在稍后介绍。

控制器的主要功能如下：

① 从主存中取出一条指令，并指出下一条指令在主存中的位置。

② 对指令进行译码或测试，产生相应的操作控制信号，以便启动规定的动作。

③ 指挥并控制CPU、主存和输入输出设备之间的数据流动方向。

运算器的主要功能如下：

① 执行所有的算术运算。

② 执行所有的逻辑运算,并进行逻辑测试。

6.1.4 CPU 的主要技术参数

CPU 品质的高低直接决定了一个计算机系统的档次,而 CPU 的主要技术参数可以反映出 CPU 的大致性能。

1. 字长

CPU 的字长是指在单位时间内同时处理的二进制数据的位数。CPU 按照其处理信息的字长可以分为:8 位 CPU、16 位 CPU、32 位 CPU 以及 64 位 CPU 等。

2. 内部工作频率

内部工作频率又称内频或主频,它是衡量 CPU 速度的重要参数,计量单位为 MHz(兆赫)、GHz(吉赫)。CPU 的主频表示在 CPU 内数字脉冲信号震荡的速度,与 CPU 实际的运算能力存在一定的关系,但还没有一个确定的公式能够定量两者的数值关系,因为 CPU 的运算速度还要看 CPU 的流水线的各方面的性能指标(缓存、指令集、CPU 的位数等),因此主频仅是 CPU 性能表现的一个方面,而不代表 CPU 的整体性能。目前,主流 CPU 的主频一般为几 GHz。

内部时钟频率的倒数是时钟周期,这是 CPU 中最小的时间元素。每个动作至少需要一个时钟周期。

以 PC 系列微处理器为例,最初的 8086 和 8088 执行一条指令平均需要 12 个时钟周期;80286 和 80386 的速度提高,每条指令大约要 4.5 个时钟周期;80486 的速度进一步提高,每条指令大约 2 个时钟周期;Pentium 具有双指令流水线,使得每个时钟周期执行 1～2 条指令;而 Pentium pro、Pentium Ⅱ/PentiumⅢ每个时钟周期可以执行 3 条或更多的指令。

3. 外部工作频率

CPU 除了主频之外,还有另一种工作频率,称为外部工作频率,它是由主板为 CPU 提供的基准时钟频率。

在早期,CPU 的内频就等于外频。例如,80486DX-33 的内频是 33MHz,它的外频也是 33MHz。也就是说,80486DX-33 以 33MHz 的速度在内部进行运算,也同样以 33MHz 的速度与外界沟通。目前,CPU 的内频越来越高,相比之下,其他设备的速度还很缓慢,所以现在外频跟内频不再只是一比一的同步关系,从而出现了所谓的内部倍频技术,导致了“倍频”的出现。内频、外频和倍频三者之间的关系是:

$$内频=外频\times倍频$$

例如,80486DX2-66 的外频是 33MHz,由于内部 2 倍频技术的关系,外频的值会自动乘上一个因数 2,而成为内频(66MHz)。到了 Pentium 时代,由于 CPU 支持多种倍频,因此在设定 CPU 的频率时,不仅要设定外频,也要指定倍频。

目前,CPU 的内频已高达数个 GHz,而外频才发展为 266MHz、400MHz 等,与 CPU 的差距很大,目前最高的倍频数甚至可达 34。例如,Intel Core i7 主频 3.4GHz,外频仅 100MHz。理论上倍频是从 1.5 一直到无限,以 0.5 为一个间隔单位。

4. 前端总线频率

前端总线(Front Side Bus)通常用 FSB 表示,它是 CPU 和外界(北桥芯片)交换数据的通道,主要连接主存、显卡等数据吞吐率高的部件,因此前端总线的数据传输能力对计算机整体性能作用很大。如果没有足够快的前端总线,再强的 CPU 也不能明显提高计算机的整体速度。

在 Pentium 4 出现之前,前端总线频率与外频是相同的,因此往往直接称前端总线频率为外频。随着计算机技术的发展,需要前端总线频率高于外频,采用了 QDR(Quad Date Rate)技术或者其他类似的技术,使得前端总线频率成为外频的 2 倍、4 倍甚至更高。

FSB 的频率采用“MHz”作为单位,PC 上主流的前端总线频率有 800MHz、1066MHz、1333MHz、1600MHz 几种,前端总线频率越高,代表着 CPU 与内存之间的数据传输率越高。

$$数据带宽=总线频率\times数据位宽\div 8$$

例如,64 位、1600MHz 的 FSB 所提供的内存带宽是 1600MHz×64b÷8=12 800MB/s=12.5GB/s。

虽然前端总线频率看起来已经很高,但与同时不断提升的内存频率、高性能显卡相比,前端总线瓶颈仍未根本改变。目前,大多数 CPU 中的 FSB 已被 QPI 总线或 DMI 总线取代,为新一代的处理器提供更快、更高效的数据带宽,FSB 的系统瓶颈问题也随之得以解决。

5. QPI 数据传输速率

快速通道互联(Quick Path Interconnect,QPI)是一种取代 FSB 的基于包传输的高速点到点连接技术。原来北桥芯片中的内存控制器集成到 CPU 内部,让 CPU 通过 PQI 总线直接和内存通信,不再通过北桥芯片组,这很明显加快了速度。

QPI 抛弃了 FSB 易混淆的单位“MHz”,而使用“GT/s”或“MT/s”,明确地表示总线实际的数据传输速率,而不是时钟频率。T/s 即 transfers per second,表示每秒数据传输的次数。QPI 总线采用的是 2∶1 比率,即实际的数据传输速率两倍于实际的总线时钟频率。QPI 的时钟频率基于 2.4 GHz、3.2GHz,则 QPI 的数据传输速率为 4.8GT/s、6.4GT/s。例如,时钟频率 2.4GHz 的 QPI 的数据传输速率是 2.4GHz×2=4.8GT/s。

一个基本的 QPI 数据包是 80 位,需要 4 次传输完成整个数据包的传输,每次传输的 20 位数据中,有 16 位为有效数据,其余 4 位用于循环冗余校验。由于 QPI 是双向的,在发送的同时也可以接收另一端传输来的数据,这样,每个 QPI 总线总带宽=每秒传输次数(即 QPI 速率)×每次传输的有效数据(即 16b/8=2B)×双向。所以 QPI 速率为 4.8GT/s 的总带宽=4.8GT/s×2B×2=19.2GB/s,QPI 速率为 6.4GT/s 的总带宽=6.4GT/s×2B×2=25.6GB/s。不难发现,目前的 QPI 比以前最宽、最快的 FSB 还要快一倍。

6. DMI 数据传输速率

DMI 是指直接媒体接口(Direct Media Interface)。它基于 PCI-Express 总线(详见第 8 章介绍),并跟随 PCI-E 总线的更新而换代。DMI 采用点对点的连接方式,时钟频率为 100MHz。

DMI 并不是新鲜事物,在没有取消 FSB 时,DMI 就是 Intel 公司开发用于北桥和南桥之间的芯片连接总线。DMI 实现了上行与下行双向数据传输率,单通道单向传输速率达到 2.5GT/s,采用 8b/10b 编码,共计 4 条通道。

随着集成电路的发展,PCI-E 控制器也被整合进了 CPU,这样一来,相当于北桥芯片的功能整个都集成到了 CPU 内部,主板上看不到北桥芯片的踪影,只剩下一个名为 PCH(平台控制中枢),它的性质类似于过去的南桥,它与 CPU 之间不需要交换太多数据,因此连接总线采用 DMI 足够了,故 FSB 被 DMI 取代。

随着 PCI-E 总线的升级 DMI 的数据传输速率在不断提高,DMI2.0 的单通道传输速率达到 5GT/s,仍采用 8b/10b 编码;DMI3.0 的单通道传输速率达到 8GT/s,采用 128b/130b 编码,有效码率高达 98.46%,相比 8b/10b 编码的 80%提高了很多。

DMI 总线带宽的计算公式:

理论最大带宽(GB/s)=(传输速率×编码率×通道数)÷8

DMI 理论最大带宽=(2.5GT/s×8/10×4)÷8=1GB/s

DMI2.0 理论最大带宽=(5GT/s×8/10×4)÷8=2GB/s

DMI3.0 理论最大带宽=(8GT/s×128/130×4)÷8=3.94GB/s

Intel CPU 通过 FSB、QPI、DMI 总线传输数据示意如图 6-3 所示。

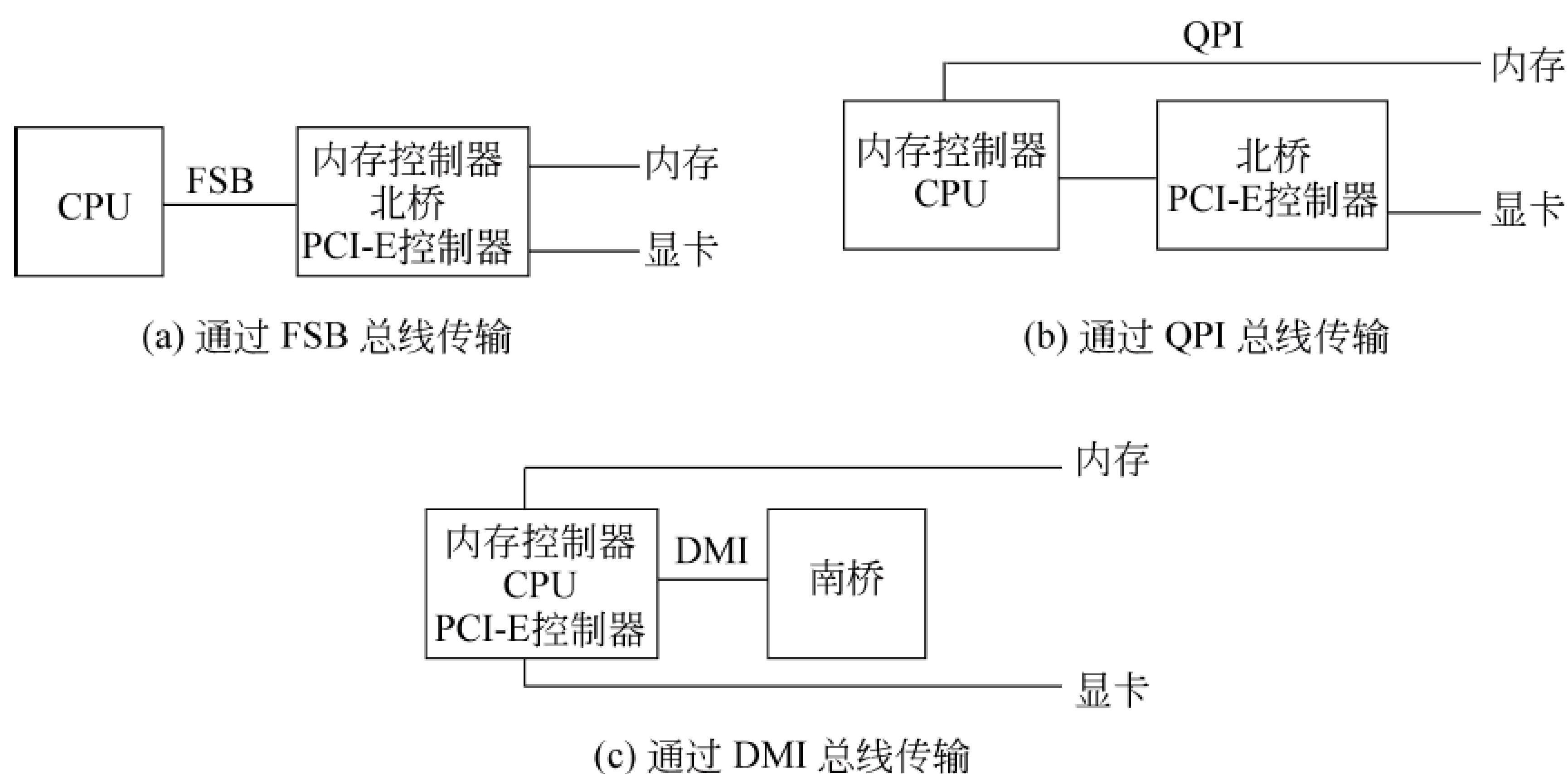


图 6-3 Intel CPU 通过 FSB、QPI、DMI 总线传输数据

7. 片内 Cache 的容量

片内 Cache 又称 CPU Cache,它的容量和工作速率对提高计算机的速度起着关键作用。CPU Cache 可以分为 L1 Cache、L2 Cache,部分高端 CPU 还具有 L3 Cache。

L1 Cache:位于 CPU 内核的旁边,是与 CPU 结合最为紧密的 CPU 缓存。一般来说,一级缓存可以分为一级数据缓存(Data Cache,D-Cache)和一级指令缓存(Instruction Cache,I-Cache)。大多数 CPU 的一级数据缓存和一级指令缓存具有相同的容量。例如,D-Cache 和 I-Cache 各为 64KB,总容量为 128KB。

L2 Cache:是影响 CPU 性能的关键因素之一,在 CPU 核心不变化的情况下,增加 L2 Cache 的容量能使性能大幅度提高,而同一核心 CPU 的高低端之分往往也是在 L2 Cache

上有差异。目前 CPU 的 L2 Cache 一般为 1MB，容量可达 4～8MB。

L3 Cache：是为读取 L2 Cache 后未命中的数据设计的一种缓存，在拥有 L3 Cache 的 CPU 中，只有约 5%的数据需要从内存中调用，这进一步提高了 CPU 的效率。L3 Cache 的容量一般从几兆字节至几十兆字节不等。

8. 工作电压

工作电压指的是 CPU 正常工作所需的电压。早期 CPU 的工作电压一般为 5V，以至于 CPU 的发热量太大，使得寿命缩短。随着 CPU 的制造工艺与内频的提高，近年来各种 CPU 的工作电压有逐步下降的趋势，以解决发热问题，目前一般台式计算机的 CPU 工作电压已低于 3V，有的已低于 2V；而便携式计算机专用 CPU 的工作电压就更低了，甚至达到 1.2V。这使得功耗大大减少，但其生产成本也大为提高。

9. 地址总线宽度

地址总线宽度决定了 CPU 可以访问的最大的物理地址空间，简单地说就是 CPU 到底能够使用多大容量的主存。例如，Pentium 有 32 位地址线，可寻址的最大容量为 2^{32} = 4096MB(4GB)，而 Itantium 有 44 位地址线，可寻址的最大容量为 2^{44} = 16TB。

10. 数据总线宽度

数据总线宽度决定了 CPU 与外部 Cache、主存以及输入输出设备之间进行一次数据传输的信息量。如果数据总线为 32 位，则每次最多可以读写主存中的 32 位；如果数据总线为 64 位，则每次最多可以读写主存中的 64 位。

数据总线和地址总线是互相独立的，数据总线宽度指明了芯片的信息传递能力，而地址总线宽度说明了芯片可以访问多少个主存单元。

11. TDP 功耗

TDP 是指热设计功耗(Thermal Design Power)。TDP 的含义是当处理器达到最大负荷的时候所释放出的热量，计量单位为瓦(W)。这是反应处理器热量释放的指标，是计算机的冷却系统必须有能力驱散的最大热量限度。

TDP 功耗可以大致反映出 CPU 的发热情况，而制约 CPU 发展的一个重要问题就是散热问题，显然发热量低的 CPU 设计有望达到更高的工作频率，目前的台式计算机 CPU 的 TDP 功耗超过 100W 基本是不可取的，比较理想的数值是低于 50W；而便携式计算机 CPU 的 TDP 功耗仅 15W。

12. 制造工艺

线宽是指芯片内电路与电路之间的距离，可以用线宽来描述制造工艺。线宽越小，就意味着芯片上包括的晶体管数目越多。Pentium Ⅱ 的线宽是 0.35μm，晶体管数达到 7.5 兆个；Pentium Ⅲ 的线宽是 0.25μm，晶体管数达到 9.5 兆个；Pentium 4 的线宽是 0.18μm，晶体管数达到 42 兆个。目前主流 CPU 的线宽 45nm、32nm、22nm 和 14nm，7nm 的制造工艺将是新一代 CPU 的发展目标。

6.2 控制器的组成和实现方法

控制器是计算机系统的指挥中心,它把运算器、存储器、输入输出设备等部件组成一个有机的整体,然后根据指令的要求指挥全机的工作。

6.2.1 控制器的基本组成

各种不同类型计算机的控制器会有不少差别,但其基本组成是相同的,图 6-4 给出了控制器的基本组成框图,控制器主要由以下几部分组成。

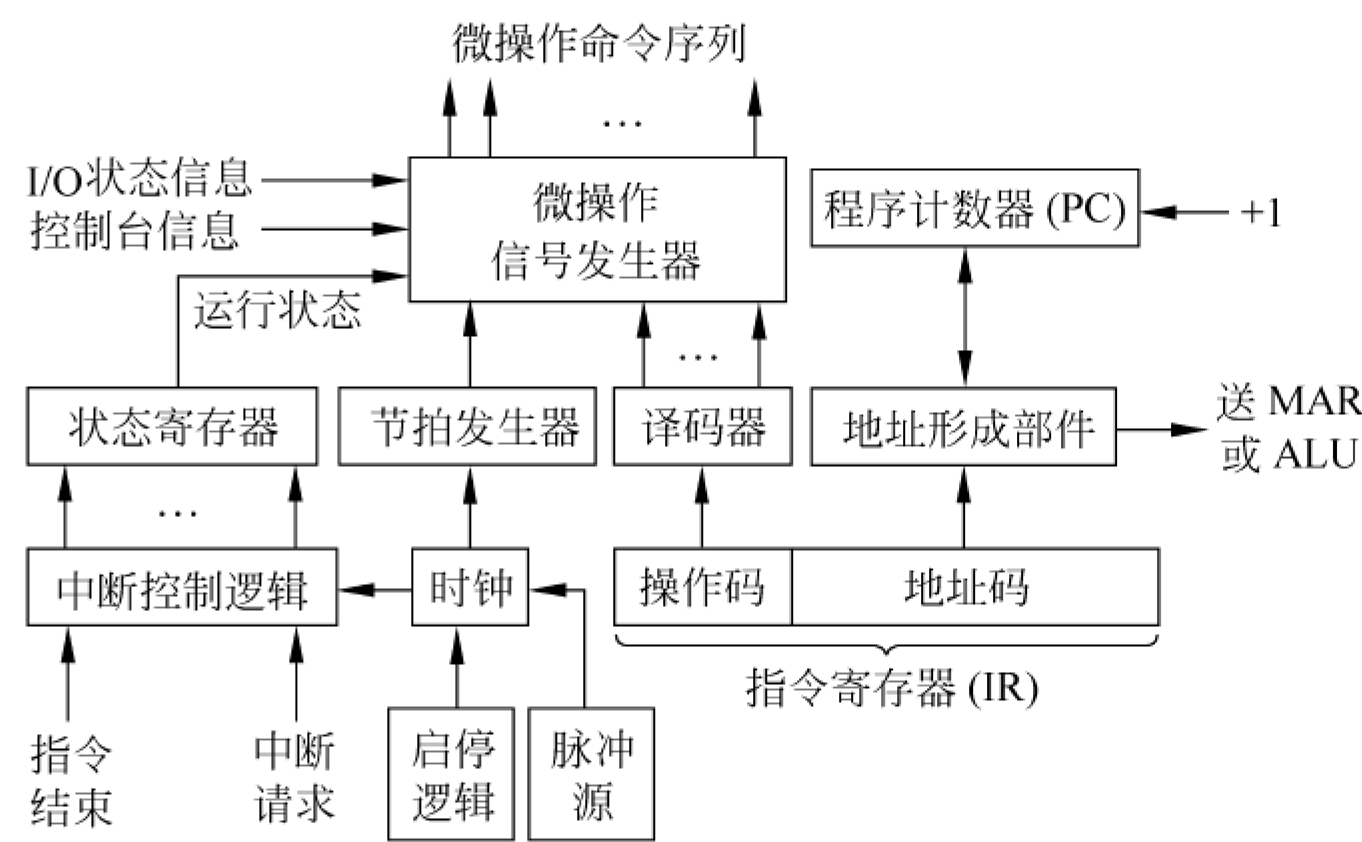


图 6-4 控制器的基本组成

1. 指令部件

指令部件的主要任务是完成取指令并分析指令。指令部件包括以下几个部分。

(1) 程序计数器

程序计数器说明见 6.1.2 节。

(2) 指令寄存器

指令寄存器说明见 6.1.2 节。

(3) 指令译码器

指令译码器又称操作码译码器或指令功能分析解释器。暂存在指令寄存器中的指令,只有在其操作码部分经过译码之后,才能识别出这是一条什么样的指令,并产生相应的控制信号提供给微操作信号发生器。

(4) 地址形成部件

地址形成部件根据指令的不同寻址方式,形成操作数的有效地址。在微型和小型机中,可以不设专门的地址形成部件,而利用运算器来进行有效地址的计算。

2. 时序部件

时序部件能产生一定的时序信号,以保证机器的各功能部件有节奏地进行信息传送、加工及信息存储。时序部件包括以下几个部分。

(1) 脉冲源

脉冲源用来产生具有一定频率和宽度的时钟脉冲信号，为整个机器提供基准信号。为使主脉冲的频率稳定，一般都使用石英晶体振荡器做脉冲源。当计算机的电源一接通，脉冲源立即按规定的频率重复发出具有一定占空比的时钟脉冲序列，直至关闭电源为止。

(2) 启停控制逻辑

只有通过启停控制逻辑将计算机启动后，主时钟脉冲才允许进入，并启动节拍信号发生器开始工作。启停控制逻辑的作用是根据计算机的需要，可靠地开放或封锁脉冲，控制时序信号的发生或停止，实现对整个机器的正确启动或停止。启停控制逻辑保证启动时输出的第一个脉冲和停止时输出的最后一个脉冲都是完整的脉冲。

(3) 节拍信号发生器

节拍信号发生器又称脉冲分配器。脉冲源产生的脉冲信号，经过节拍信号发生器后产生出各个机器周期中的节拍信号，用以控制计算机完成每一步微操作。

3. 微操作信号发生器

一条指令的取出和执行可以分解成很多最基本的操作，这种最基本的不可再分割的操作称为微操作。微操作信号发生器也称为控制单元(CU)。不同的机器指令具有不同的微操作序列。

4. 中断控制逻辑

中断控制逻辑是用来控制中断处理的硬件逻辑。有关中断的问题将在第9章中专门介绍。

6.2.2 控制器的硬件实现方法

控制器的核心是微操作信号发生器(控制单元CU)，图6-5是反映控制单元外特性的框图。微操作控制信号是由指令部件提供的译码信号、时序部件提供的时序信号和被控制功能部件所反馈的状态及条件综合形成的。

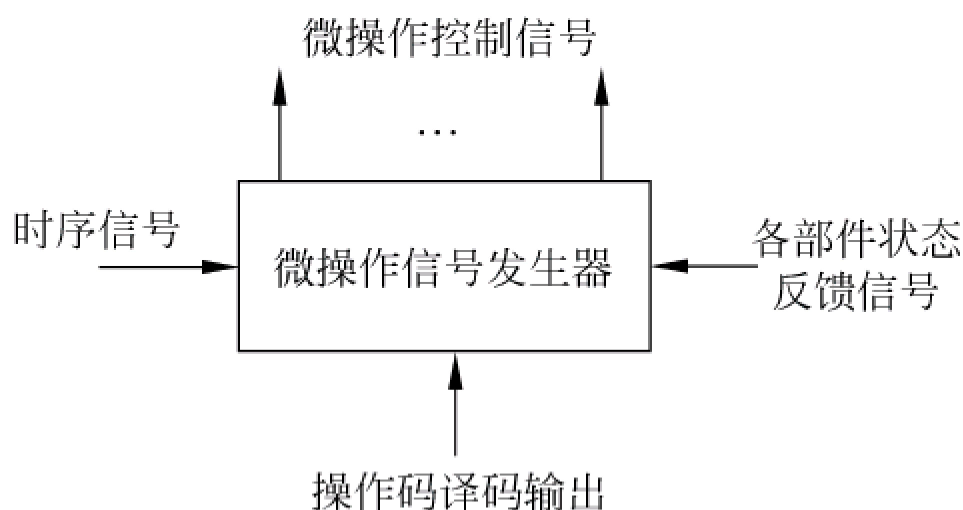


图6-5 控制单元外特性

控制单元的输入包括时序信号、机器指令操作码、各部件状态反馈信号等，输出的微操作控制信号又可细分为CPU内的控制信号和送至主存或外设的控制信号。根据产生微操作控制信号的方式不同，控制器可分为组合逻辑型、存储逻辑型、组合逻辑与存储逻辑结合型3种，它们的根本区别在于控制单元的实现方法不同，而控制器中的其他部分基本上是大同小异的。

1. 组合逻辑型控制器

这种控制器称为常规控制器或硬连线控制器,是采用组合逻辑技术来实现的,其控制单元是由门电路组成的复杂树形网络。这种方法是分立元件时代的产物,以使用最少器件数和取得最高操作速度为设计目标。

组合逻辑型控制器的最大优点是速度快。但是,组合逻辑型控制器控制单元的结构不规整,使得设计、调试、维修较困难,难以实现设计自动化;一旦控制单元构成之后,要想增加新的控制功能是不可能的。因此,它受到微程序控制器的强烈冲击。目前,仅有一些巨型机和 RISC 机为了追求高速度仍采用组合逻辑型控制器。

2. 存储逻辑型控制器

这种控制器称为微程序控制器,是采用存储逻辑来实现的,也就是把微操作信号代码化,使每条机器指令转化成为一段微程序并存入一个专门的存储器(控制存储器)中,微操作控制信号由微指令产生。

微程序控制器的设计思想和组合逻辑设计思想截然不同。它具有设计规整、调试、维修以及更改、扩充指令方便的优点,易于实现自动化设计,但是,由于它增加了一级控制存储器,所以指令的执行速度比组合逻辑型控制器慢。

3. 组合逻辑和存储逻辑结合型控制器

这种控制器称为可编程逻辑阵列(PLA)控制器,是吸收前两种方法的设计思想来实现的。PLA 控制器实际上也是一种组合逻辑型控制器,但它又与常规的组合逻辑型控制器的硬联结构不同;它是可编程序的,某一微操作控制信号由 PLA 的某一输出函数产生。

PLA 控制器是组合逻辑技术和存储逻辑技术结合的产物,克服了两者的缺点,是一种较有前途的方法。

6.3 时序系统与控制方式

由于计算机高速地进行工作,每一个动作的时间是非常严格的,不能有任何差错。时序系统是控制器的心脏,其功能是为指令的执行提供各种定时信号。

6.3.1 时序系统

1. 指令周期和机器周期

指令周期是指从取指令、分析取数到执行完该指令所需的全部时间。由于各种指令的操作功能不同,有的简单,有的复杂,因此各种指令的指令周期不尽相同。

机器周期又称 CPU 周期。通常把一个指令周期划分为若干个机器周期,每个机器周期完成一个基本操作。一般机器的 CPU 周期有取指周期、取数周期、执行周期和中断周期等。所以有:

$$指令周期 = i \times 机器周期$$

不同的指令周期中所包含的机器周期数差别可能很大。一般情况下，一条指令所需的最短时间为两个机器周期：取指周期和执行周期。

通常，每个机器周期都有一个与之对应的周期状态触发器。机器运行在不同的机器周期时，其对应的周期状态触发器被置“1”。显然，在机器运行的任何时刻只能处于一种周期状态，因此，有一个且仅有一个触发器被置“1”。

由于CPU内部的操作速度较快，而CPU访问主存所花的时间较长，所以许多计算机系统往往以主存的工作周期(存取周期)为基础来规定CPU周期，以便两者的工作能配合协调。CPU访问主存也就是一次总线传送，故在微型计算机中称为总线周期。

2. 节拍

在一个机器周期内，要完成若干个微操作。这些微操作有的可以同时执行，有的需要按先后次序串行执行。因而应把一个机器周期分为若干个相等的时间段，每一个时间段对应一个电位信号，称为节拍电位信号。

节拍的宽度取决于CPU完成一次微操作(如ALU一次正确的运算、寄存器间的一次传送等)的时间。

由于不同的机器周期内需要完成的微操作内容和个数是不同的，因此，不同机器周期内所需要的节拍数也不相同。节拍的选取一般有以下几种方法。

(1) 统一节拍法

以最复杂的机器周期为准定出节拍数，每一个节拍时间的长短也以最繁的微操作作为标准。这种方法采用统一的、具有相等时间间隔和相同数目的节拍，使得所有的机器周期长度都是相等的，因此称为定长CPU周期。

(2) 分散节拍法

按照机器周期的实际需要安排节拍数，需要多少节拍就发出多少节拍，这样可以避免浪费，提高时间利用率。由于各机器周期长度不同，故称为不定长CPU周期。

(3) 延长节拍法

在照顾多数机器周期要求的情况下，选取适当的节拍数作为基本节拍。如果在某个机器周期内统一的节拍数无法完成该周期的全部微操作，则可以延长一或两个节拍。

(4) 时钟周期插入法

在一些微型计算机中，时序信号中不设置节拍，而直接使用时钟周期信号。一个机器周期中含有若干个时钟周期，时钟周期的数目取决于机器周期内完成微操作数目的多少及相应功能部件的速度。一个机器周期的基本时钟周期数确定之后，还可以不断插入等待时钟周期。例如，8086的一个总线周期(即机器周期)中包含4个基本时钟周期T_1～T_4，在T_3和T_4之间可以插入任意个等待时钟周期T_W，以等待速度较慢的存储部件或外部设备完成读或写操作，如图6-6所示。

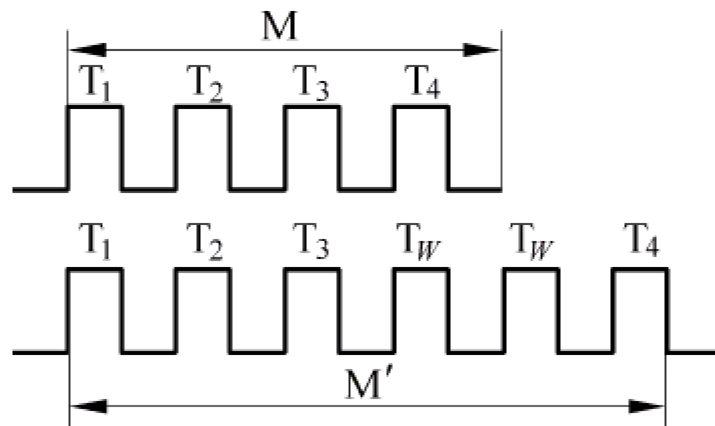


图6-6 时钟周期的插入

3. 工作脉冲

在节拍中执行的有些微操作需要同步定时脉冲，如将稳定的运算结果打入寄存器，又如

机器周期状态切换等。为此,在一个节拍内常常设置一个或几个工作脉冲作为各种同步脉冲的来源。工作脉冲的宽度只占节拍电位宽度的$\frac{1}{n}$,并处于节拍的末尾部分,以保证所有的触发器都能可靠、稳定地翻转。

在只设置机器周期和时钟周期的微型计算机中,一般不再设置工作脉冲,因为时钟周期既可以作为电位信号,其前、后沿又可以作为脉冲触发信号。

4. 多级时序系统

图 6-7 为小型计算机每个指令周期中常采用的机器周期、节拍、工作脉冲三级时序系统。图中每个机器周期 M 中包括 4 个节拍 $T_1 \sim T_4$,每个节拍内有一个脉冲 P。在机器周期间、节拍电位间、工作脉冲间既不允许有重叠交叉,也不允许有空隙,应该是一个接一个的准确连接。

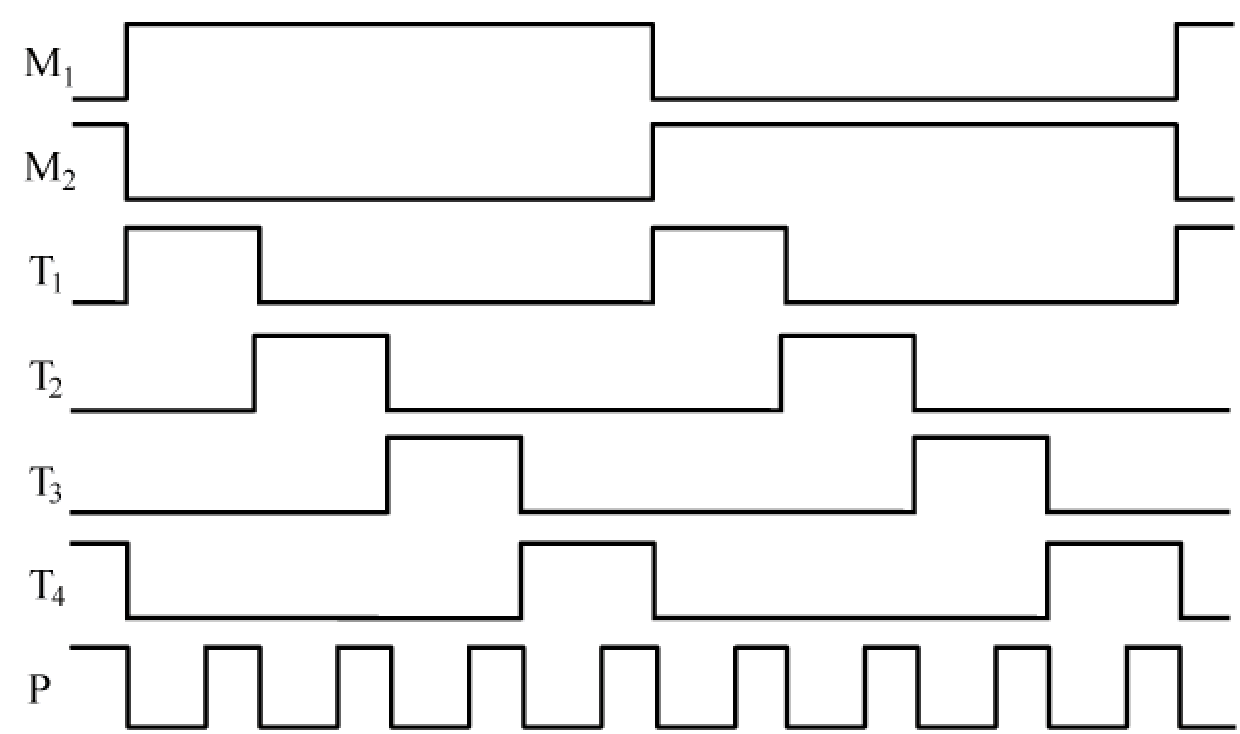


图 6-7　三级时序系统

微型计算机中常用的时序系统与小型机的略有不同,称为时钟周期时序系统。一个指令周期包含若干个机器周期,一个机器周期又包含若干个时钟周期。

5. 节拍电位和工作脉冲的时间配合关系

在计算机中,节拍电位和工作脉冲所起的控制作用是不同的。电位信号是信息的载体,即控制信号,它在数据通路传输中起着开门或关门的作用;工作脉冲则作为打入脉冲加在触发器的脉冲输入端,起到定时触发的作用。通常,触发器使用电位—脉冲工作方式,节拍电位控制信息送到 D 触发器的 D 输入端,工作脉冲送到 CP 输入端。节拍电位和工作脉冲配合关系如图 6-8 所示。

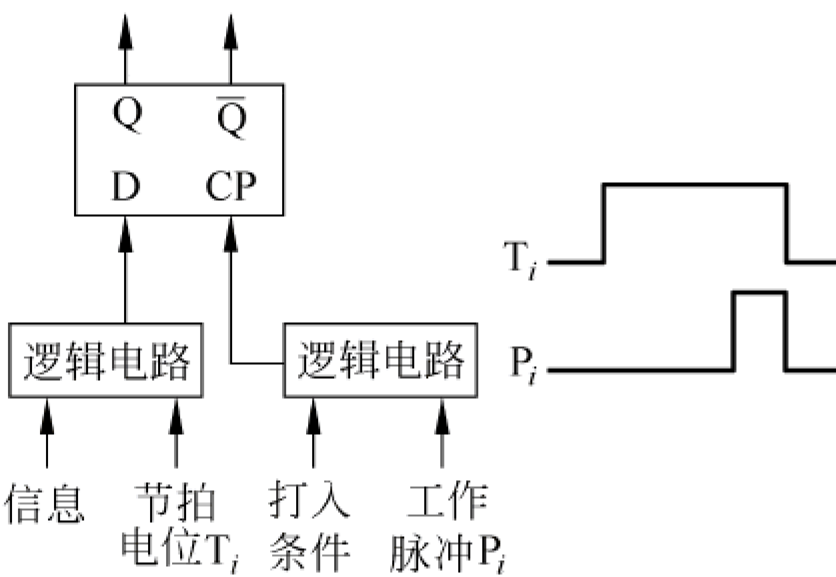


图 6-8　节拍电位和工作脉冲的配合关系

6.3.2 控制方式

CPU 的控制方式可以分为以下 3 种。

1. 同步控制方式

同步控制方式即固定时序控制方式，各项操作都由统一的时序信号控制，在每个机器周期中产生统一数目的节拍电位和工作脉冲。由于不同的指令，操作时间长短不一致。同步控制方式应以最复杂指令的操作时间作为统一的时间间隔标准。

同步控制方式设计简单，容易实现；但是，对于许多简单指令来说这种控制方式会有较多的空闲时间，造成较大数量的时间浪费，从而影响了指令的执行速度。

在同步控制方式中，各指令所需的时序由控制器统一发出，所有微操作都与时钟同步，所以又称为集中控制方式或中央控制方式。

2. 异步控制方式

异步控制方式即可变时序控制方式，各项操作不采用统一的时序信号控制，而根据指令或部件的具体情况决定，需要多少时间就占用多少时间。

这是一种“应答”方式，各操作之间的衔接是由“结束－起始”信号来实现的。由前一项操作已经完成的“结束”信号，或由下一项操作的“准备好”信号来作为下一项操作的起始信号，在未收到“结束”或“准备好”信号之前不开始新的操作。例如，存储器读操作时，CPU 向存储器发一个读命令(起始信号)，启动存储器内部的时序信号，以控制存储器读操作，此时 CPU 处于等待状态。当存储器操作结束后，存储器向 CPU 发出 MFC(结束信号)，以此作为下一项操作的起始信号。

异步控制采用不同时序，没有时间上的浪费，因而提高了机器的效率，但是控制比较复杂。

由于这种控制方式没有统一的时钟，而是由各功能部件本身产生各自的时序信号自我控制，所以又称为分散控制方式或局部控制方式。

3. 联合控制方式

联合控制方式是同步控制和异步控制相结合的方式。实际上现代计算机中几乎没有完全采用同步或完全采用异步的控制方式，大多数是采用联合控制方式。通常的设计思想是：在功能部件内部采用同步方式或以同步方式为主的控制方式，在功能部件之间采用异步方式。

例如，在一般小型和微型计算机中，CPU 内部基本时序采用同步方式，按多数指令的需要设置节拍数。对于某些复杂指令如果节拍数不够，可采取延长节拍等方法，以满足指令的要求。当 CPU 通过总线向主存或其他外设交换数据时，就转入异步方式。CPU 只需给出起始信号，主存和外设按自己的时序信号去安排操作；一旦操作结束，则向 CPU 发结束信号，以便 CPU 再安排它的后继工作。

6.3.3 指令运行的基本过程

一条指令运行过程可以分为 3 个阶段：取指令阶段、分析取数阶段和执行阶段。

1. 取指令阶段

取指令阶段完成的任务是将现行指令从主存中取出来并送至指令寄存器中，具体的操

作如下:

① 将程序计数器(PC)中的内容送至存储器地址寄存器(MAR),并送地址总线(AB)。

② 由控制单元(CU)经控制总线(CB)向存储器发读命令。

③ 从主存中取出的指令通过数据总线(DB)送到存储器数据寄存器(MDR)。

④ 将MDR的内容送至指令寄存器(IR)中。

⑤ 将PC的内容递增,为取下一条指令做好准备。

以上这些操作对任何一条指令来说都是必须要执行的操作,所以称为公共操作。完成取指阶段任务的时间称为取指周期,图6-9给出了取指周期中CPU各部分的工作流程。

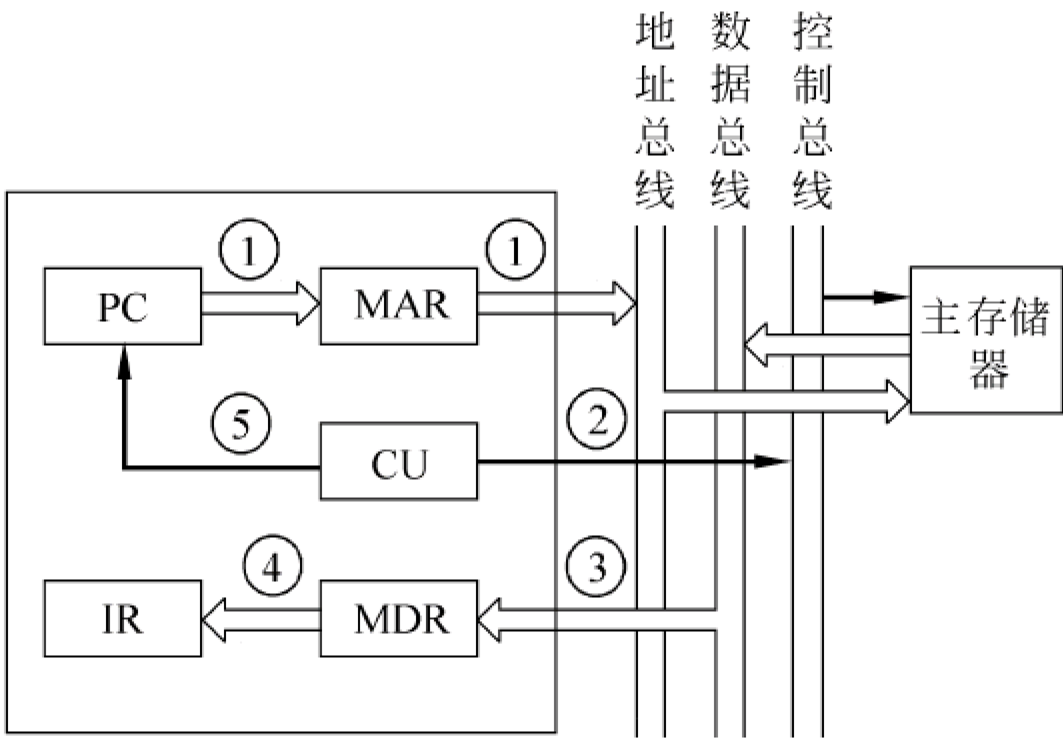


图6-9 取指周期中CPU各部分的工作流程

2. 分析取数阶段

取出指令后,指令译码器(ID)可识别和区分出不同的指令类型。此时计算机进入分析取数阶段,以获取操作数。由于各条指令功能不同,寻址方式也不同,所以分析取数阶段的操作是各不相同的。

对于无操作数指令,只要识别出是哪条具体的指令即可直接转至执行阶段,所以无须进入分析取数阶段。而对于带操作数指令,为读取操作数,首先要计算出操作数的有效地址。如果操作数在通用寄存器中,则不需要再访问主存;如果操作数在主存中,则要到主存中去取数。对于不同的寻址方式,有效地址的计算方法是不同的,有时要多次访问主存才能取出操作数来(间接寻址)。另外,单操作数指令和双操作数指令由于需要的操作数的个数不同,分析取数阶段的操作也不同。

完成分析取数阶段任务的时间又可以细分为间址周期、取数周期等。

3. 执行阶段

执行阶段完成指令规定的各种操作,形成稳定的运算结果,并将其存储起来。完成执行阶段任务的时间称为执行周期。

计算机的基本工作过程就是取指令、取数、执行指令,然后再取下一条指令……如此周而复始,直至遇到停机指令或外来的干预为止。

6.3.4 指令的微操作序列

控制器在实现一条指令的功能时，总要把每条指令分解成一系列时间上先后有序的最基本、最简单的微操作，即微操作序列。微操作序列是与 CPU 的内部数据通路密切相关的，不同的数据通路就有不同的微操作序列。

假设某计算机的数据通路如图 6-10 所示。规定各部件用大写字母表示，字母加下标 in 表示该部件的接收控制信号，实际上就是该部件的输入开门信号；字母加下标 out 表示该部件的发送控制信号，实际上就是该部件的输出开门信号。例如，MAR_{in}、PC_{out} 等就是这类微操作信号。下面分析具体指令发出的微操作控制信号。

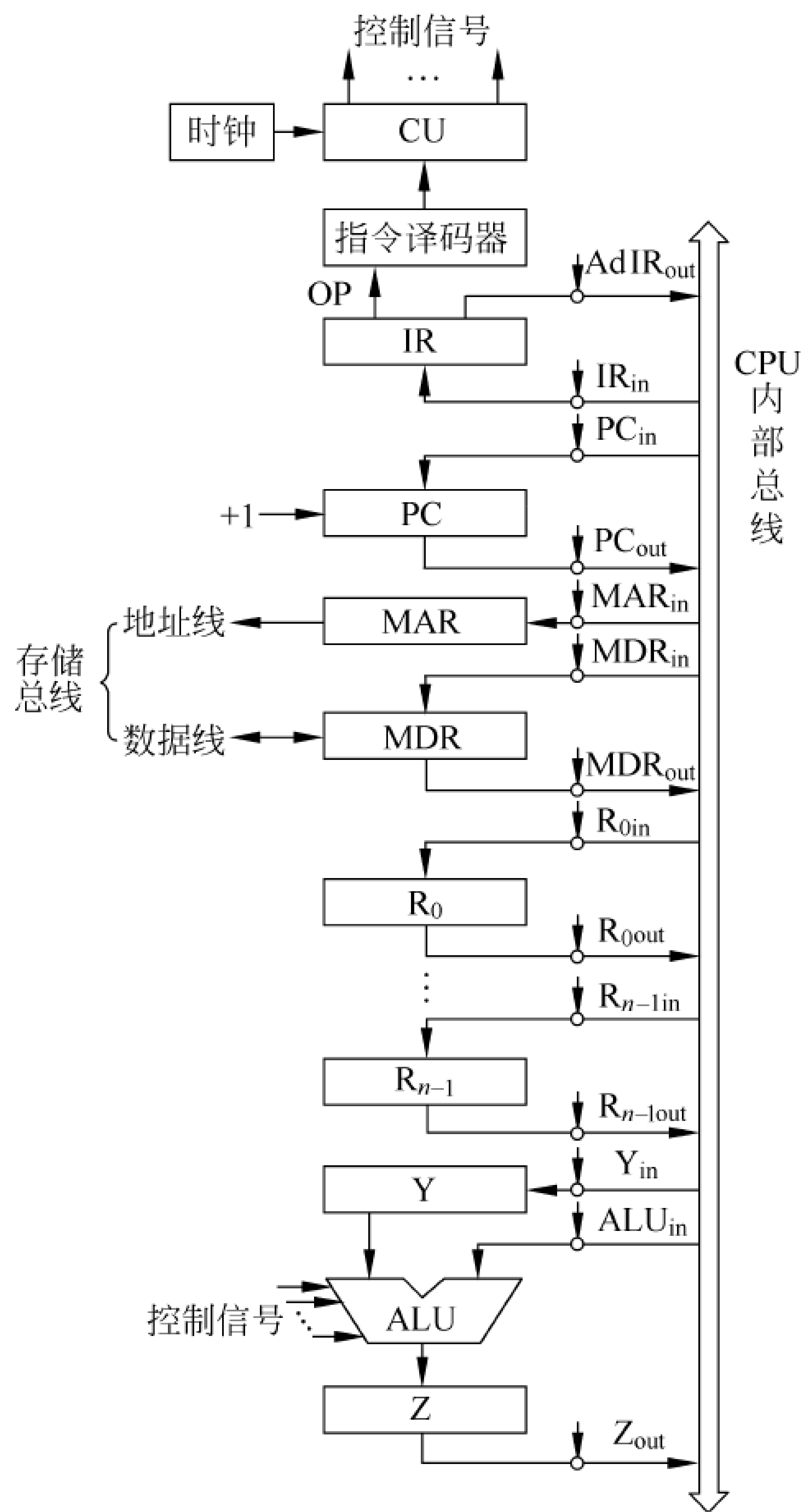


图 6-10 CPU 的数据通路

1. 加法指令 ADD@R_0，R_1

这条指令完成的功能是把 R_0 的内容作为地址送到主存以取得一个操作数，再与 R_1 中的内容相加，最后将结果送回主存中。即实现：

$$((R_0))+(R_1)\rightarrow(R_0)$$

(1) 取指周期

取指周期完成的微操作序列是公共的操作,与具体指令无关。

① PC_{out} 和 MAR_{in} 有效,完成 PC 的内容(即指令地址)经 CPU 内部总线送至 MAR 的操作,记作(PC)→MAR。

② 通过控制总线(图中未画出)向主存发读命令,记作 Read。

③ 存储器通过数据总线将 MAR 所指单元的内容(指令)送至 MDR,记作 M(MAR)→MDR。

④ MDR_{out} 和 IR_{in} 有效,将 MDR 的内容送至指令寄存器,记作(MDR)→IR。至此,指令被从主存中取出,其操作码字段经过指令译码器开始控制 CU。

⑤ 使 PC 内容加 1,记作(PC)+1→PC。

(2) 取数周期

取数周期要完成从主存中取操作数的任务,此时另一个操作数已放在寄存器 R_1 中。

① R_{0out} 和 MAR_{in} 有效,完成将被加数地址送至 MAR 的操作,记作(R_0)→MAR。

② 向主存发读命令,记作 Read。

③ 存储器通过数据总线将 MAR 所指单元的内容(即数据)送至 MDR,同时 MDR_{out} 和 Y_{in} 有效,记作 M(MAR)→MDR→Y。

(3) 执行周期

执行周期完成加法运算的任务,并将结果写回主存。

① R_{1out} 和 ALU_{in} 有效,同时 CU 向 ALU 发“ADD”控制信号,使 R_1 的内容和 Y 的内容相加,结果送寄存器 Z,记作(R_1)+(Y)→Z。

② Z_{out} 和 MDR_{in} 有效,将运算结果送 MDR,记作(Z)→MDR。

③ 向主存发写命令,记作 Write。

2. 转移指令 JC A

这是一条条件转移指令,若上次运算结果有进位(C=1),就转移;若上次运算结果无进位(C=0),就顺序执行下一条指令。设 A 为位移量,转移地址等于 PC 的内容加位移量。相应的微操作序列如下。

(1) 取指周期

与上条指令的微操作序列完全相同。

(2) 执行周期

如果有进位(C=1),则完成(PC)+A→PC 的操作,否则跳过以下几步。

① PC_{out} 和 Y_{in} 有效,记作(PC)→Y(C=1)。

② Ad IR_{out} 和 ALU_{in} 有效,同时 CU 向 ALU 发“ADD”控制信号,使 IR 中的地址码字段 A 和 Y 的内容相加,结果送寄存器 Z,记作 Ad(IR)+(Y)→Z(C=1)。

③ Z_{out} 和 PC_{in} 有效,将转移地址送入 PC,记作(Z)→PC(C=1)。

6.4 微程序控制原理

微程序设计技术的实质是将程序设计技术和存储技术相结合,即用程序设计的思想方法来组织操作控制逻辑,将微操作控制信号按一定规则进行信息编码(代码化),形成控制字

(微指令),再把这些微指令按时间先后排列起来构成微程序,存放在一个只读的控制存储器中。

6.4.1 微程序控制的基本概念

1. 微程序设计的提出与发展

微程序设计的概念和原理最早是由英国剑桥大学的 M. V. Wilkes 教授于 1951 年提出来的。他在《设计自动化计算机的最好方法》中指出:一条机器指令可以分解为许多基本的微命令序列,并且首先把这种思想用于计算机控制器的设计。但是,由于当时还不具备制造专门存放微程序的控制存储器的技术,所以在十几年时间内实际上并未真正使用。直到 1964 年,IBM 公司在 IBM360 系列机上成功地采用了微程序设计技术,解决了指令系统的兼容问题。20 世纪 70 年代以来,由于 VLSI 技术的发展,推动了微程序设计技术的发展和应用,目前大多数计算机都采用微程序设计技术。

2. 基本术语

(1) 微命令和微操作

前面已经提到,一条机器指令可以分解成一个微操作序列,这些微操作是计算机中最基本的、不可再分解的操作。在微程序控制的计算机中,将控制部件向执行部件发出的各种控制命令称为微命令,它是构成控制序列的最小单位。例如,打开或关闭某个控制门的电位信号、某个寄存器的打入脉冲等。因此,微命令是控制计算机各部件完成某个基本微操作的命令。

微命令和微操作是一一对应的。微命令是微操作的控制信号,微操作是微命令的操作过程。

微命令有兼容性和互斥性之分。兼容性微命令是指那些可以同时产生,共同完成某一些微操作的微命令;而互斥性微命令是指在机器中不允许同时出现的微命令。兼容和互斥都是相对的,一个微命令可以和一些微命令兼容,和另一些微命令互斥。对于单独一个微命令,谈论其兼容和互斥都是没有意义的。

(2) 微指令、微地址

微指令是指控制存储器中的一个单元的内容,即控制字,是若干个微命令的集合。存放控制字的控制存储器的单元地址就称为微地址。

一条微指令通常至少包含两部分信息:

① 操作控制字段,又称微操作码字段,用以产生某一步操作所需的各微操作控制信号。

② 顺序控制字段,又称微地址码字段,用以控制产生下一条要执行的微指令地址。

微指令有垂直型和水平型之分。垂直型微指令接近于机器指令的格式,每条微指令只能完成一个基本微操作;水平型微指令则具有良好的并行性,每条微指令可以完成较多的基本微操作。

(3) 微周期

从控制存储器中读取一条微指令并执行相应的微命令所需的全部时间称为微周期。

(4) 微程序

一系列微指令的有序集合就是微程序。每一条机器指令都对应一个微程序。

注意：微程序和程序是两个不同的概念。微程序是由微指令组成的，用于描述机器指令，微程序实际上是机器指令的实时解释器，是由计算机的设计者事先编制好并存放在控制存储器中的，一般不提供给用户。对于程序员来说，计算机系统中微程序一级的结构和功能是透明的，无须知道。而程序最终由机器指令组成，是由软件设计人员事先编制好并存放在主存或辅存中的。所以说，微程序控制的计算机涉及两个层次：一是机器语言或汇编语言程序员所看到的传统机器层，包括机器指令、工作程序和主存储器；二是机器设计者看到的微程序层，包括微指令、微程序和控制存储器。

6.4.2 微指令编码法

微指令可以分成操作控制字段和顺序控制字段两部分。这里所说的微指令编码法指的就是操作控制字段的编码方法。各类计算机从各自的特点出发，设计了各种各样的微指令编码法。例如，大型机强调速度，要求译码过程尽量快；微型机和小型机则更多地注意经济性，要求更大限度地缩短微指令字长；而中型机介于这两者之间，兼顾速度和价格，要求在保证一定速度的情况下，能尽量缩短微指令字长。下面从基本原理出发，对几种基本的微指令编码方法进行讨论。

1. 直接控制法(不译码法)

直接控制法顾名思义是操作控制字段中的各位分别可以直接控制计算机，无须进行译码。在这种形式的微指令字中，操作控制字段的每一个独立的二进制位代表一个微命令，该位为“1”表示这个微命令有效，为“0”则表示这个微命令无效。每个微命令对应并控制数据通路中的一个微操作。

这种方法结构简单，并行性强，操作速度快，但是微指令字太长。若微命令的总数为 N 个，则微指令字的操作控制字段就要有 N 位。在某些计算机中，微命令的总数可能会多达三、四百个，甚至更多，这使微指令的长度达到难以接受的地步。另外，在 N 个微命令中，有许多是互斥的，不允许并行操作，将它们安排在一条微指令中是毫无意义的，只会使信息的利用率下降。所以，这种方法在复杂的系统中很少单独采用，往往与其他编码方法混合起来使用。

2. 最短编码法

直接控制法使微指令字过长，而最短编码法则走向另一个极端，使得微指令字最短。这种方法将所有的微命令统一编码，每条微指令只定义一个微命令。若微命令的总数为 N，操作控制字段的长度为 L，则最短编码法应满足下列关系式：

$$L \geqslant \log_2 N$$

最短编码法的微指令字长最短，但要通过一个微命令译码器译码以后才能得到需要的微命令。微命令数目越多，译码器就越复杂。这种方法在同一时刻只能产生一个微命令，不能充分利用机器硬件所具有的并行性，使得机器指令对应的微程序变得很长，而且对于某些要求在同一时刻同时动作的组合性微操作将无法实现。因此，这种方法也只能与其他方法混合使用。

3. 字段编码法

这是前述两种编码法的一个折中的方法,既具有两者的优点,又克服了它们的缺点。这种方法将操作控制字段分为若干个小段,每段内采用最短编码法,段与段之间采用直接控制法。这种方法又可进一步分为字段直接编码法和字段间接编码法。

(1) 字段直接编码法

图 6-11 为字段直接编码法的微指令结构,各字段都可以独立地定义本字段的微命令,而和其他字段无关,因此又称为显式编码或单重定义编码方法。这种方法缩短了微指令字,因此得到了广泛的应用。

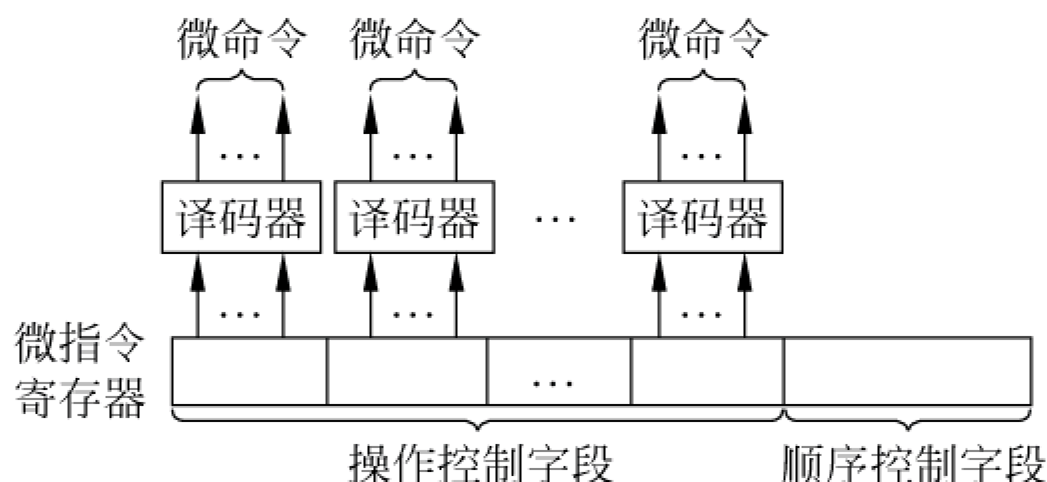


图 6-11 字段直接编码法

(2) 字段间接编码法

字段间接编码法是在字段直接编码法的基础上,用来进一步缩短微指令字长的方法。间接编码的含义是,一个字段的某些编码不能独立地定义某些微命令,而需要与其他字段的编码来联合定义,因此又称为隐式编码或多重定义编码方法,如图 6-12 所示。

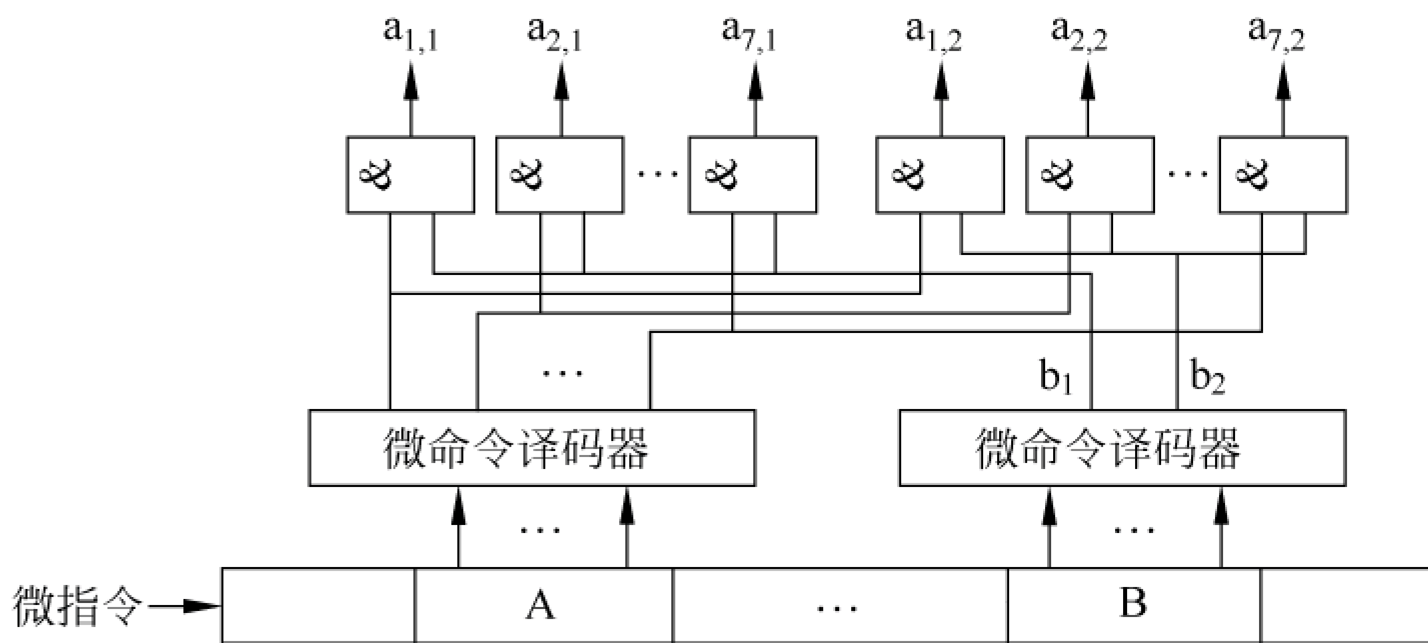


图 6-12 字段间接编码法

图 6-12 中字段 A(假设 3 位)所产生的微命令还要受到字段 B 的控制。当字段 B 发出 b_1 微命令时,字段 A 与其合作产生 $a_{1,1}$、$a_{2,1}$、…、$a_{7,1}$ 中的一个微命令;而当字段 B 发出 b_2 微命令时,字段 A 与其合作产生 $a_{1,2}$、$a_{2,2}$、…、$a_{7,2}$ 中的另一个微命令。这种方法进一步减少了微指令的长度,但通常可能会削弱微指令的并行控制能力,且译码电路相应地较复杂,因此,它只作为字段直接编码法的一种补充。

字段编码法中操作控制字段的分段并非是任意的,必须要遵循如下原则:

① 把互斥性的微命令分在同一段内,兼容性的微命令分在不同段内。这样不仅有助于提高信息的利用率,缩短微指令字长,而且有助于充分利用硬件所具有的并行性,加快执行的速度。

② 应与数据通路结构相适应。

③ 每个小段中包含的信息位不能太多,否则将增加译码线路的复杂性和译码时间。

④ 一般每个小段还要留出一个状态,表示本字段不发出任何微命令。因此,当某字段的长度为3位时,最多只能表示7个互斥的微命令,通常用000表示不操作。

例如,运算器的输出控制信号有直传、左移、右移、半字交换4个。这4个微命令是互斥的。它们可以安排在同一字段编码内。同样,存储器的读写命令也是一对互斥的微命令。还有像A→C、B→C(假设A、B、C都是寄存器)这样一类的微命令也是互斥的微命令,不允许它们在同一时刻出现。

假设某计算机共有256个微命令,如果采用直接控制法,微指令的操作控制字段就要有256位;而如果采用最短编码法,操作控制字段只需要8位就可以了。如果采用字段直接编码法,若4位为一个段,每段可表示15个互斥的微命令,则操作控制字段只需要72位,分成18个段,在同一时刻可以并行发出18个不同的微命令。

除上述几种基本的编码方法外,另外还有一些常见的编码技巧,例如可采用微指令译码与部分机器指令译码的复合控制、微地址参与解释微指令译码等。对于实际机器的微指令系统,通常同时采用几种不同的编码方法。例如,在一条微指令中,可以有些位采用直接控制法,有些字段采用直接编码法,另一些字段采用间接编码法。总之,要尽量减少微指令字长,增强微操作的并行性,提高机器的控制性能并降低成本。

6.4.3 微程序控制器的组成和工作过程

1. 微程序控制器的基本组成

图6-13给出了一个微程序控制器基本结构的简化框图,在图中主要画出了微程序控制器比组合逻辑控制器多出的部件,包括控制存储器、微指令寄存器、微地址形成部件和微地址寄存器等。

(1) 控制存储器(CM)

这是微程序控制器的核心部件,用来存放微程序,其性能(包括容量、速度、可靠性等)与计算机的性能密切相关。

(2) 微指令寄存器(μIR)

用来存放从CM中取出的微指令,它的位数同微指令字长相等。

(3) 微地址形成部件

用来产生初始微地址和后继微地址,以保证微指令的连续执行。

(4) 微地址寄存器(μMAR)

它接受微地址形成部件送来的微地址,为在CM中读取微指令做准备。

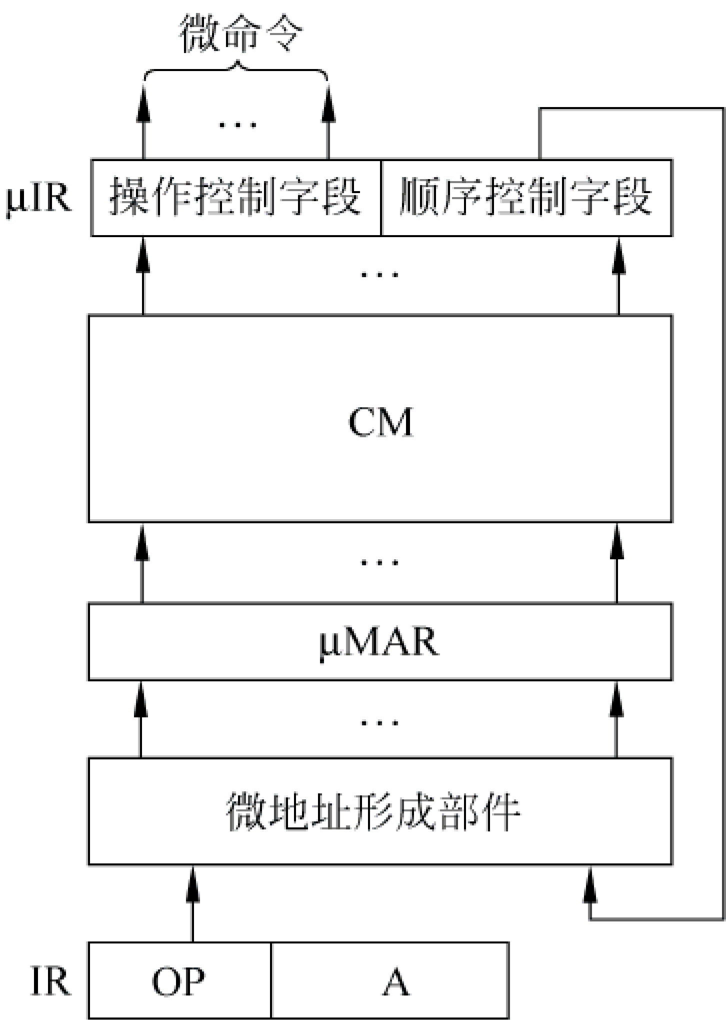


图6-13 微程序控制器的基本结构

2. 微程序控制器的工作过程

微程序控制器的工作过程实际上就是在微程序控制器的控制下计算机运行机器指令的

过程,这个过程可以描述如下:

① 执行取指令公共操作。取指令的公共操作通常由一个取指微程序来完成,这个取指微程序也可能仅由一条微指令组成。具体的执行是:在机器开始运行时,自动将取指微程序的入口微地址送入 μMAR,并从 CM 中读出相应的微指令送入 μIR。微指令的操作控制字段产生有关的微命令,用来控制计算机实现取机器指令的公共操作。取指微程序的入口地址一般为 CM 的 0 号单元,当取指微程序执行完后,从主存中取出的机器指令就已存入指令寄存器中。

② 由机器指令的操作码字段通过微地址形成部件产生该机器指令所对应的微程序的入口地址,并送入 μMAR。

③ 从 CM 中逐条取出对应的微指令并执行之。

④ 执行完对应于一条机器指令的一个微程序后又回到取指微程序的入口地址,继续第①步,以完成取下一条机器指令的公共操作。

以上是一条机器指令的运行过程,如此周而复始,直到整个程序执行完毕为止。

3. 机器指令对应的微程序

通常,一条机器指令对应一个微程序。由于任何一条机器指令的取指令操作都是相同的,因此可以将取指令操作抽出来编成一个独立的微程序,这个微程序只负责将指令从主存中取出送至指令寄存器。此外,也可以编出对应间址周期的微程序和中断周期的微程序。这样,控制存储器中的微程序个数应等于指令系统中的机器指令数再加上对应取指、间址和中断周期等公用的微程序数。若指令系统中具有 n 种机器指令,则控制存储器中的微程序数至少有 $n+1$ 个。

6.4.4 微程序入口地址的形成

当公用的取指微程序从主存中取出机器指令之后,由机器指令的操作码字段指出各个微程序的入口地址(初始微地址)。这是一种多分支(或多路转移)的情况。由机器指令的操作码转换成初始微地址的方式主要有 3 种。

1. 一级功能转换

如果机器指令操作码字段的位数和位置固定,可以直接使操作码与入口地址码的部分位相对应。例如,某计算机系统有 16 条机器指令,指令操作码由 4 位二进制数表示,分别为 0000、0001、…,1111。现以字母 θ 表示操作码,令微程序的入口地址为 θ11B,例如,MOV 指令的操作码为 0000,则 MOV 指令的微程序入口地址为 000011B;ADD 指令的操作码为 0001,则 ADD 指令的微程序入口地址为 000111B……由此可见,相邻两个微程序的入口地址相差 4 个单元,如图 6-14 所示。也就是说,每个微程序最多可以由 4 条微指令组成,如果不足 4 条就让有关单元空闲着。

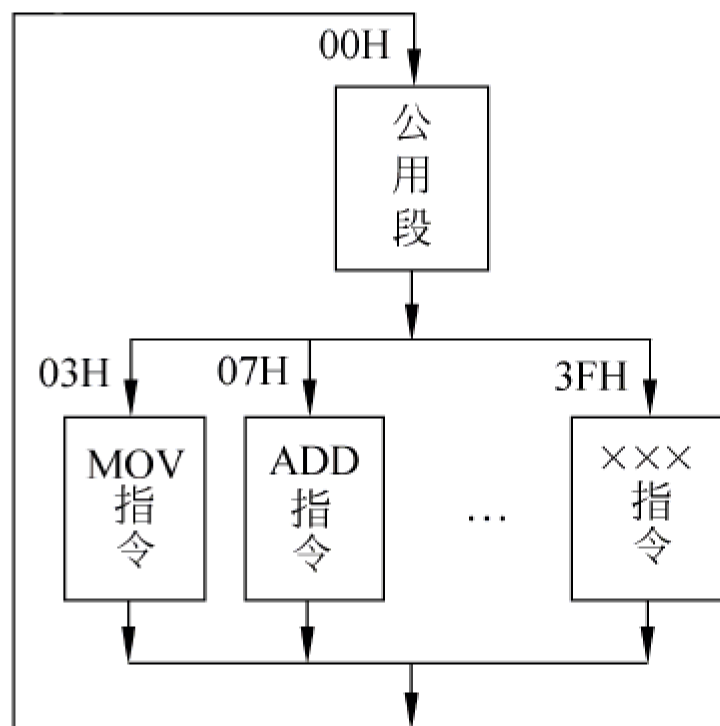


图 6-14 指令操作码与微程序入口地址

2. 二级功能转换

当同类机器指令的操作码字段的位数和位置固定,而不同类机器指令的操作码字段的位数和位置不固定时,就不能再采用一级功能转换的方法。所谓二级功能转换是指第一次先按指令类型标志转移,以区分出指令属于哪类,如是单操作数指令,还是双操作数指令等。因为每类机器指令中操作码字段的位数和位置是固定的,所以第二次即可按操作码区分出具体是哪条指令,以便找出相应微程序的入口微地址。

3. 通过PLA电路实现功能转换

当机器指令的操作码位数和位置都不固定时,可以采用PLA电路将每条机器指令的操作码翻译成对应的微程序入口地址。这种方法对于变长度、变位置的操作码显得更有效,而且转换速度较快。

6.4.5 后继微地址的形成

找到初始微地址之后,可以开始执行微程序,每条微指令执行完毕都要根据要求形成后继微地址。后继微地址的形成方法对微程序编制的灵活性影响很大,它主要有两个基本类型:增量方式和断定方式。

1. 增量方式(顺序—转移型微地址)

这种方式和机器指令的控制方式很类似,它也有顺序执行、转移和转子之分。顺序执行时,后继微地址就是现行微地址加上一个增量(通常为"1");转移或转子时,由微指令的顺序控制字段产生转移微地址。因此,在微程序控制器中应当有一个微程序计数器(μPC)。为了降低成本,一般情况下都是将微地址寄存器(μMAR)改为具有计数功能的寄存器,以代替μPC。

增量方式的优点是简单,易于掌握,编制微程序容易,每条机器指令所对应的一段微程序一般安排在CM的连续单元中;其缺点是这种方式不能实现两路以上的并行微程序转移,因而不利于提高微程序的执行速度。

2. 断定方式

断定方式的后继微地址可由微程序设计者指定,或者根据微指令所规定的测试结果直接决定后继微地址的全部或部分值。

这是一种直接给定与测试断定相结合的方式,其顺序控制字段一般由两部分组成:非测试段和测试段。

① 非测试段:可由设计者指定,一般是微地址的高位部分,用来指定后继微地址在CM中的某个区域内。

② 测试段:根据有关状态的测试结果确定其地址值,一般对应微地址的低位部分。这相当于在指定区域内断定具体的分支。所依据的测试状态可能是指定的开关状态、指令操作码和状态字等。

测试段如果只有一位,则微地址将产生2个分支;若有两位,则最多可产生4个分支;以

此类推，测试段为 n 位最多可产生 2^n 个分支。

断定方式的优点是实现多路并行转移容易，有利于提高微程序的执行效率和执行速度，且微程序在 CM 中不要求必须连续存放；其缺点是后继微地址的生成机构比较复杂。

6.4.6 微程序设计

1. 微程序设计方法

在实际进行微程序设计时，应考虑尽量缩短微指令字长，减少微程序长度，提高微程序的执行速度。这几项指标是互相制约的，应当全面地进行分析和权衡。

（1）水平型微指令及水平型微程序设计

水平型微指令是指一次能定义并能并行执行多个微命令的微指令。它的并行操作能力强，效率高，灵活性强，执行一条机器指令所需微指令的数目少，执行时间短；但是，微指令字较长，增加了控存的横向容量，同时微指令和机器指令的差别很大，设计者只有熟悉了数据通路，才有可能编制出理想的微程序，一般用户不易掌握。由于水平型微程序设计是面对微处理器内部逻辑控制的描述，所以把这种微程序设计方法称为硬方法。

（2）垂直型微指令及垂直型微程序设计

垂直型微指令是指一次只能执行一个微命令的微指令。它的并行操作能力差，一般只能实现一个微操作，控制一两个信息传送通路，效率低，执行一条机器指令所需的微指令数目多，执行时间长；但是，微指令与机器指令很相似，所以容易掌握和利用，编程比较简单，不必过多地了解数据通路的细节，且微指令字较短。由于垂直型微程序设计是面向算法的描述，所以把这种微程序设计方法称为软方法。

（3）混合型微指令

综合上述两者特点的微指令称为混合型微指令，它具有不太长的微指令字，又具有一定的并行控制能力，可高效地实现机器的指令系统。

2. 微指令的运行方式

运行一条微指令的过程与运行机器指令的过程很类似。第一步将微指令从控存中取出，称为取微指令；对于垂直型微指令，还应包括微操作码的译码时间。第二步执行微指令所规定的各个操作。微指令的运行方式可分为串行和并行两种方式。

（1）串行方式

在这种方式里，取微指令和执行微指令是顺序进行的，在一条微指令取出并执行之后，才能取下一条微指令。图 6-15 是微指令串行运行方式的时序图。

一个微周期里，在取微指令阶段，CM 工作，数据通路等待；而在执行微指令阶段，CM 空闲，数据通路工作。

串行方式的微周期较长，但控制简单，形成后继微地址所用的硬件设备较少。

（2）并行方式

为了提高微指令的执行速度，可以将取微指令和执行微指令的操作重叠起来，从而缩短微周期。因为这两个操作是在两个完全不同的部件中执行的，所以这种重叠是完全可行的。

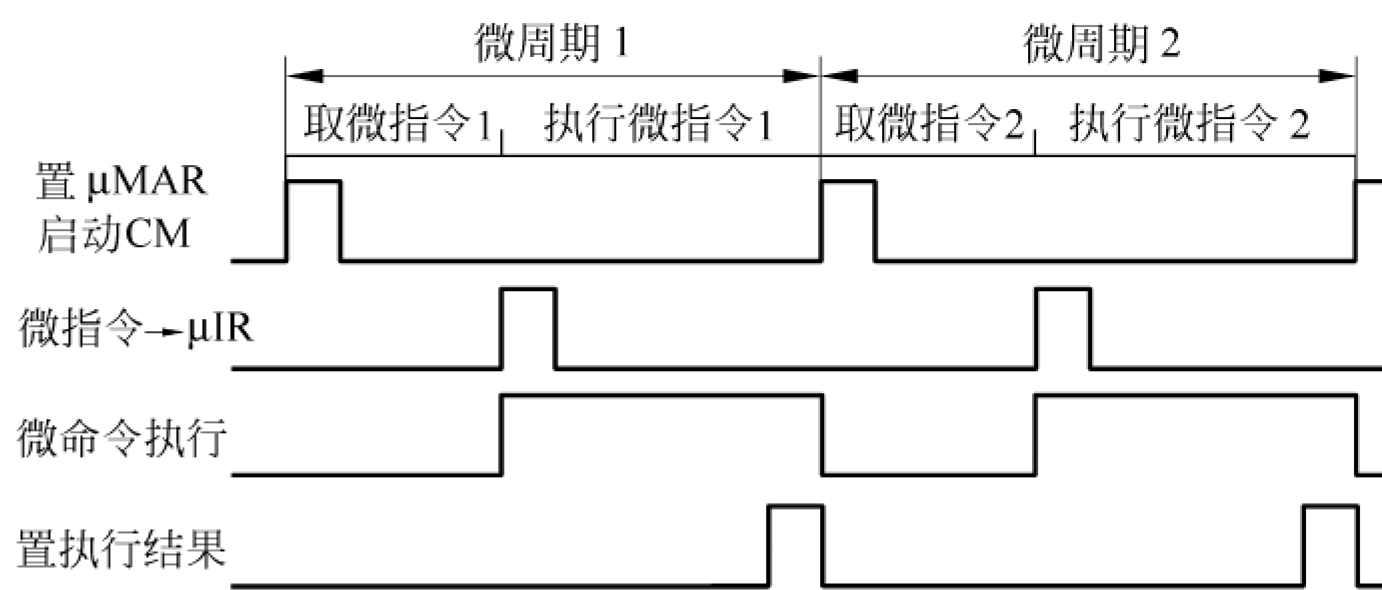


图 6-15　微指令的串行运行方式时序图

在执行本条微指令的同时，预取下一条微指令。假设取微指令的时间比执行微指令的时间短，就以较长的执行时间作为微周期，并行方式的时序如图 6-16 所示。

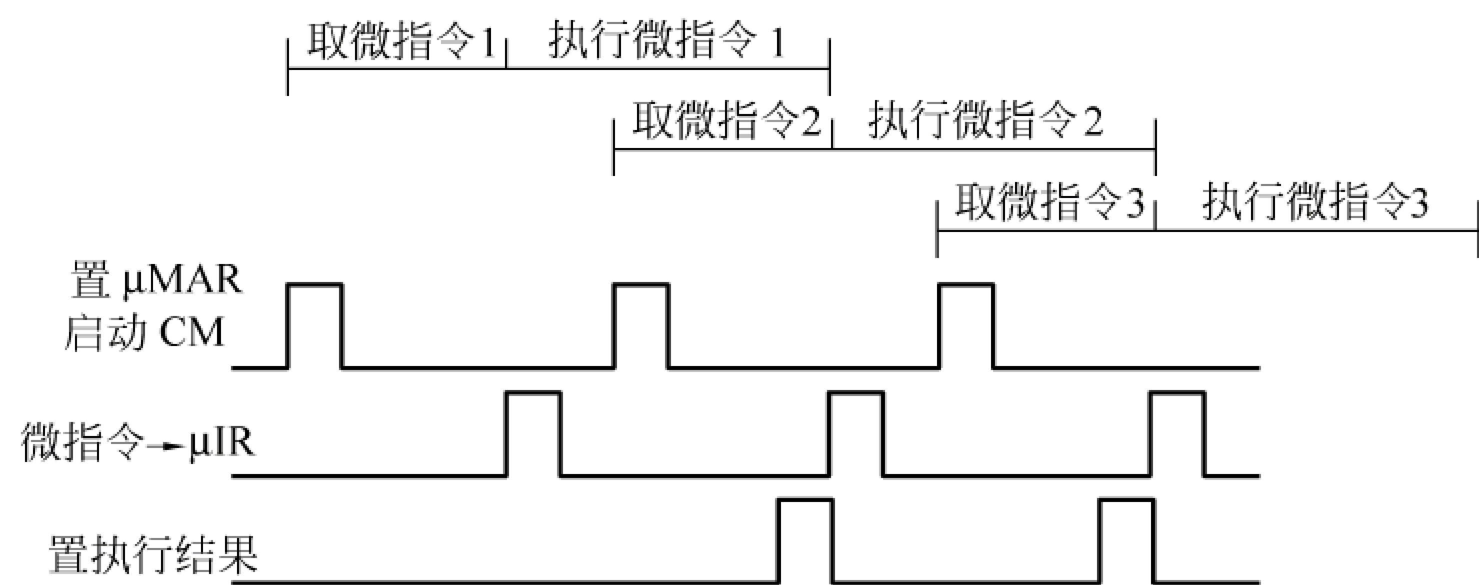


图 6-16　微指令的并行运行方式时序图

由于执行本条微指令与预取下一条微指令是同时进行的，若遇到某些需要根据本条微指令处理结果而进行条件转移的微指令，就不能并行地取出来。最简单的办法就是延迟一个微周期再取微指令。

除以上两种控制方式外，还有串、并行混合方式，即当待执行的微指令地址与现行微指令处理无关时，采用并行方式；当其受现行微指令操作结果影响时，则采用串行方式。

3. 微程序仿真

所谓微程序仿真，一般是指用一台计算机的微程序去模仿另一台计算机的指令系统，使本来不兼容的计算机之间具有程序兼容的能力。用来进行仿真的计算机称为宿主机，被仿真的计算机称为目标机。

假设 M_1 为宿主机，M_2 为目标机，在 M_1 机上要能使用 M_2 的机器语言编制程序并执行，就要求 M_1 的主存储器和控制存储器中除含有 M_1 的有关程序外，还要包含 M_2 的有关程序，如图 6-17 所示。

M_1 提供两种工作方式，本机方式和仿真方式。在本机方式时，M_1 通过本机微程序解释执行本机的程序；在仿真方式时，M_1 通过仿真微程序解释执行 M_2 的程序。

宿主机主存储器	宿主机控制存储器
本机程序	解释本机指令的微程序
目标机程序	
DOS	解释目标机指令的微程序（仿真微程序）
仿真支持软件	

图 6-17　系统仿真时宿主机的主存和控存

4. 动态微程序设计

通常,对应于一台计算机的指令系统有一系列固定的微程序。当微程序设计好之后,一般不允许改变而且也不便于改变,这样的设计叫做静态微程序设计。若一台计算机能根据不同应用目标的要求改变微程序,则这台计算机就具有动态微程序设计功能。

动态微程序设计的出发点是为了使计算机能更灵活、更有效地适应于各种不同的应用目标。例如,在不改变硬件结构的前提下,如果计算机配备了两套可供切换的微程序,一套是用来实现科学计算的指令系统,另一套是用来实现数据处理的指令系统,这样该计算机就能根据不同的应用需要随时改变和切换相应的微程序,以保证高效率地实现科学计算或数据处理。

动态微程序设计需要可写控制存储器(WCS)的支持,否则难以改变微程序的内容。由于动态微程序设计要求对计算机的结构和组成非常熟悉,所以这类改变微程序的方案也是由计算机的设计人员实现的。

5. 用户微程序设计

用户微程序设计是指用户可借助于可写控制存储器进行微程序设计,通过本机指令系统中保留的供扩充指令用的操作码或未定义的操作码,来定义用户扩充指令,然后编写扩充指令的微程序,并存入可写控存。这样用户可以如同使用本机原来的指令一样去使用扩充指令,从而大大提高计算机系统的灵活性和适应性。但是,事实上真正由用户来编写微程序是很困难的。

6.5 控制单元的设计

前面几节介绍了控制器的基本功能和 CPU 的总体结构,为了加深对这些内容的理解,这节将以一个简单的 CPU 为例来讨论控制器中控制单元的设计。为了突出重点,减少篇幅,故选择的 CPU 模型比较简单,指令系统中仅具有最常见的基本指令和寻址方式,在逻辑结构、时序安排、操作过程安排等方面尽量规整、简单,使初学者比较容易掌握,以帮助大家建立整机概念。

6.5.1 简单的 CPU 模型

控制单元的主要功能是根据需要发出各种不同的微操作控制信号。微操作控制信号是与 CPU 的数据通路密切相关的,图 6-18 给出了一个单累加器结构的简单 CPU 模型。

图 6-18 中 MAR 和 MDR 分别直接与地址总线和数据总线相连。考虑到从存储器取出的指令或有效地址都先送至 MDR 再送至 IR,故这里省去 IR 送至 MAR 的数据通路,凡是需从 IR 送至 MAR 的操作均由 MDR 送至 MAR 代替。

计算机中有一运行标志触发器 G,当 G=1 时,表示机器运行;当 G=0 时,表示停机。

这个 CPU 的指令系统中包含下列指令。

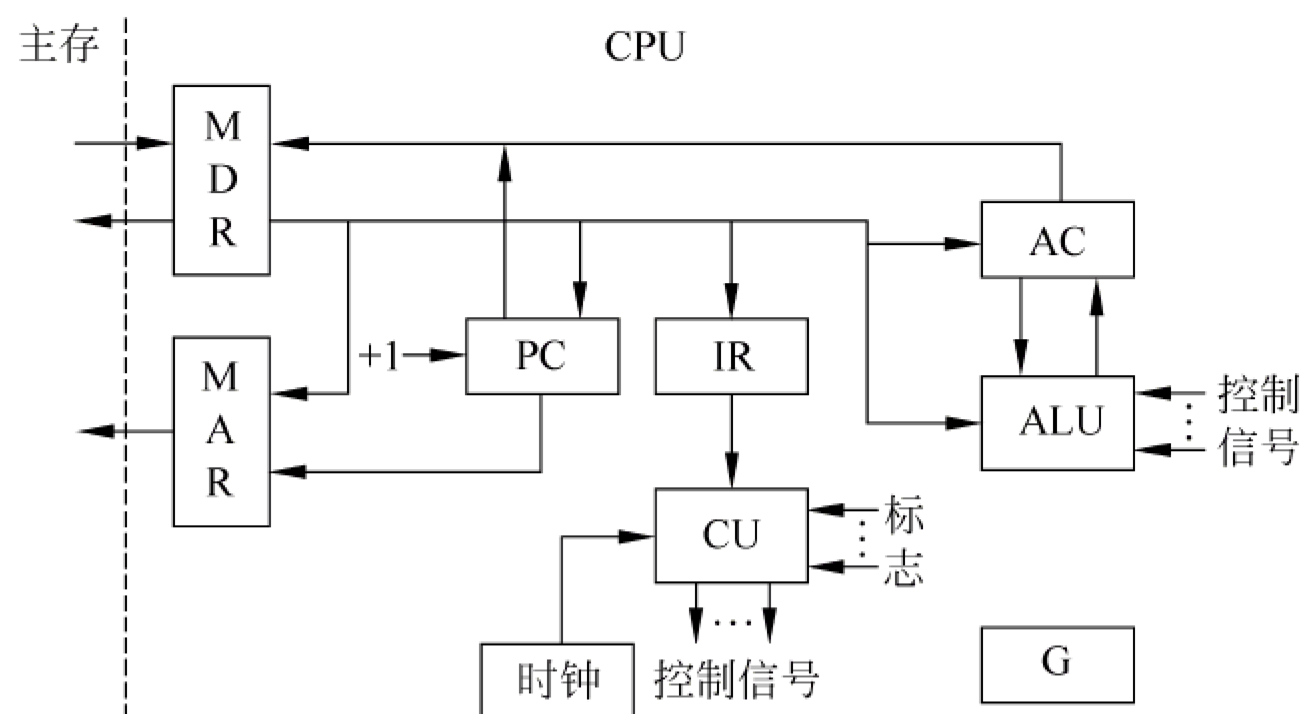


图 6-18　一个简单的 CPU 模型

1. 非访存指令

非访存指令在执行周期不访问存储器。

(1) 清除累加器指令 CLA

该指令完成清除累加器操作,记作 0→AC。

(2) 累加器取反指令 COM

该指令完成累加器内容取反,结果送累加器的操作,记作$(\overline{AC})$→AC。

(3) 累加器加 1 指令 INC

该指令完成累加器内容+1,结果送累加器的操作,记作(AC)+1→AC。

(4) 算术右移一位指令 SHR

该指令完成累加器内容算术右移一位的操作,记作 R(AC)→AC,AC_0→AC_0。

(5) 循环左移一位指令 CSL

该指令完成累加器内容循环左移一位的操作,记作 L(AC)→AC,AC_0→AC_n。

(6) 停机指令 STP

将运行标志触发器置"0",记作 0→G。

注意:累加寄存器 AC 共 $n+1$ 位,其中 AC_0 为最高位(符号位),AC_n 为最低位。AC_0→AC_0 表示算术右移时符号位保持不变。

2. 访存指令

访存指令在执行周期需访问主存储器。

(1) 加法指令 ADD

该指令完成累加器内容与对应主存单元的内容相加,结果送累加器的操作,记作(AC)+(MDR)→AC。

(2) 减法指令 SUB

该指令完成累加器内容与对应主存单元的内容相减,结果送累加器的操作,记作(AC)−(MDR)→AC。

(3) 与指令 AND

该指令完成累加器内容与对应主存单元的内容相与,结果送累加器的操作,记作

(AC)∧(MDR)→AC。

(4) 取数指令 LDA

该指令将对应主存单元的内容取至累加器中，记作(MDR)→AC。

(5) 存数指令 STA

该指令将累加器的内容存于对应主存单元中，记作(AC)→MDR。

3. 转移指令

转移指令在执行周期也不访问主存储器。

(1) 无条件转移指令 JMP

该指令完成将指令的地址码部分(即转移地址)送至 PC 的操作，记作(MDR)→PC。

(2) 零转移指令 JZ

该指令根据上一条指令运行的结果决定下一条指令的地址，若运算结果为零(标志位 Z=1)，则指令的地址码部分(即转移地址)送至 PC，否则程序按原顺序执行。由于在取指阶段已完成了(PC)+1→PC，所以当运算结果不为零时，就按取指阶段形成的 PC 执行，记作 $Z \cdot (MDR) + \overline{Z} \cdot (PC) \rightarrow PC$。

(3) 负转移指令 JN

若结果为负(标志位 N=1)，则指令的地址码部分送至 PC，否则程序按原顺序执行。记作 $N \cdot (MDR) + \overline{N} \cdot (PC) \rightarrow PC$。

(4) 进位转移指令 JC

若结果有进位(标志位 C=1)，则指令的地址码部分送至 PC，否则程序按原顺序执行。记作 $C \cdot (MDR) + \overline{C} \cdot (PC) \rightarrow PC$。

上述 3 类指令的指令周期如图 6-19 所示，其中访存指令又被细分为直接访存和间接访存两种。

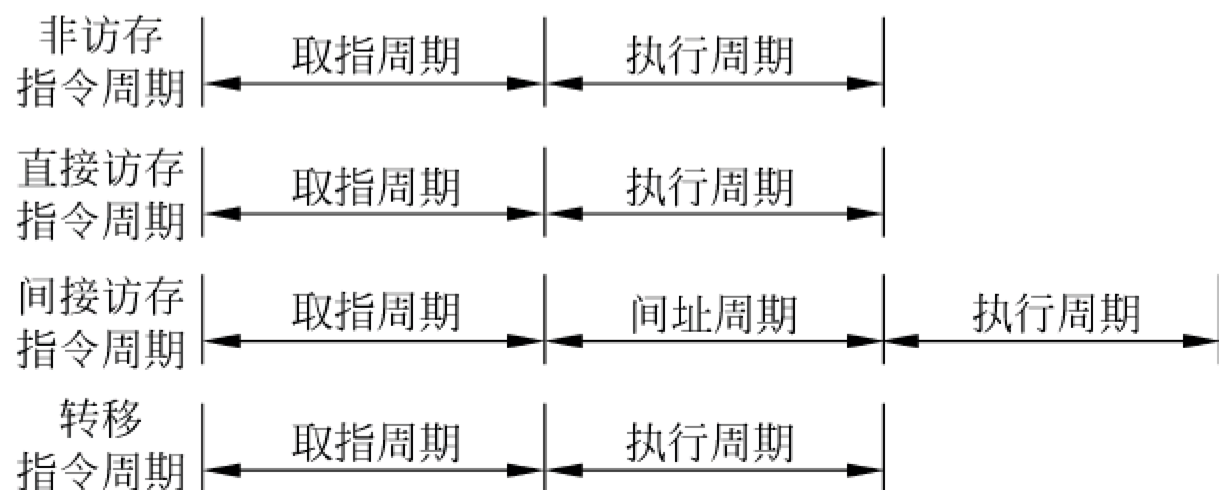


图 6-19　3 类指令的指令周期

在简单的 CPU 模型中，把一个完整的指令周期分为取指、间址、执行和中断 4 个机器周期。这 4 个机器周期中都有 CPU 访存操作，只是访存的目的不同。取指周期是为了取指令，间址周期是为了取有效地址，执行周期是为了取操作数(当指令为访存指令时)，中断周期是为了保存程序断点。这 4 个周期又可称为 CPU 工作周期，为了区别它们，在 CPU 内可设置 4 个标志触发器，如图 6-20 所示。哪个触发器处于“1”状态，就表示机器正处于哪个周期运行。因此，同一时刻有一个且仅有一个触发器处于“1”状态。

图 6-20 所示的 FE、IND、EX 和 INT 分别对应取指、间址、执行和中断 4 个周期，它们分别由 1→FE、1→IND、1→EX 和 1→INT 4 个信号控制。

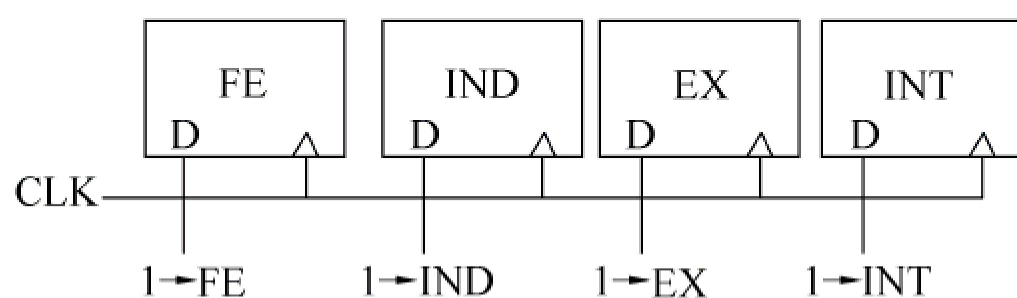


图 6-20　CPU 工作周期的标志

6.5.2　组合逻辑控制单元设计

1. 微操作的节拍安排

假设机器采用同步控制,每个机器周期包括3个节拍,安排微操作节拍时应注意以下几点:

① 有些微操作的次序是不容改变的,故安排微操作节拍时必须注意微操作的先后顺序。

② 凡是被控制对象不同的微操作,若能在一个节拍内执行,应尽可能安排在同一个节拍内,以节省时间。

③ 如果有些微操作所占的时间不长,应该将它们安排在一个节拍内完成,并且允许这些微操作有先后次序。

(1) 取指周期微操作的节拍安排

取指周期的操作是公操作,其完成的任务已在前面进行过描述,在此不再重复,这些操作可以安排在3个节拍中完成。

T_0　　(PC)→MAR,Read

T_1　　M(MAR)→MDR,(PC)+1→PC

T_2　　(MDR)→IR

考虑到指令译码时间较短,可将指令译码 OP(IR)→ID 也安排在 T_2 节拍内。

(2) 间址周期微操作的节拍安排

间址周期完成取操作数有效地址的任务,具体操作如下:

① 将指令的地址码部分(形式地址)送至存储器地址寄存器,记作(MDR)→MAR。

② 向主存发读命令,启动主存读操作,记作 Read。

③ 将 MAR 所指的主存单元中的内容(有效地址)经数据总线读至 MDR,记作 M(MAR)→MDR。

④ 将有效地址送至存储器地址寄存器(MAR),记作(MDR)→MAR。此操作在有些机器中可省略。

这些操作可以安排在以下3个节拍中完成:

T_0　　(MDR)→MAR,Read

T_1　　M(MAR)→MDR

T_2　　(MDR)→MAR

(3) 执行周期微操作的节拍安排

① 非访存指令。

非访存指令在执行周期只有一个微操作,按同步控制的原则,此操作可安排在 T_0～T_2 的任一节拍内,其余节拍空。

- 清零指令 CLA:

T_0

T_1

T_2　　$0 \to AC$

- 取反指令 COM：

T_0

T_1

T_2　　$(\overline{AC}) \to AC$

- 加 1 指令 INC：

T_0

T_1

T_2　　$(AC)+1 \to AC$

- 算术右移指令 SHR：

T_0

T_1

T_2　　$R(AC) \to AC, AC_0 \to AC_0$

- 循环左移指令 CSL：

T_0

T_1

T_2　　$L(AC) \to AC, AC_0 \to AC_n$

- 停机指令 STP：

T_0

T_1

T_2　　$0 \to G$

② 访存指令。

- 加法指令 ADD X：

T_0　　$(MDR) \to MAR$, Read

T_1　　$M(MAR) \to MDR$

T_2　　$(AC)+(MDR) \to AC$（该操作实际包括$(AC) \to ALU$，$(MDR) \to ALU$，$+$，$ALU \to AC$）

- 减法指令 SUB X：

T_0　　$(MDR) \to MAR$, Read

T_1　　$M(MAR) \to MDR$

T_2　　$(AC)-(MDR) \to AC$

- 与指令 AND X：

T_0　　$(MDR) \to MAR$, Read

T_1　　$M(MAR) \to MDR$

T_2　　$(AC) \wedge (MDR) \to AC$

- 取数指令 LDA X：

T_0　　$(MDR) \to MAR$, Read

T_1　　M(MAR)→MDR

T_2　　(MDR)→AC

• 存数指令 STA X：

T_0　　(MDR)→MAR

T_1　　(AC)→MDR，Write

T_2　　MDR→M(MAR)

③ 转移类指令。

• 无条件转移 JMP X：

T_0

T_1

T_2　　(MDR)→PC

• 结果为零转 JZ X：

T_0

T_1

T_2　　$Z \cdot (MDR)+\overline{Z} \cdot (PC) \rightarrow PC$

• 结果有进位转 JC X：

T_0

T_1

T_2　　$C \cdot (MDR)+\overline{C} \cdot (PC) \rightarrow PC$

• 结果为负转 JN X：

T_0

T_1

T_2　　$N \cdot (MDR)+\overline{N} \cdot (PC) \rightarrow PC$

2. 组合逻辑设计步骤

组合逻辑设计控制单元时，首先根据上述微操作的节拍安排，列出微操作命令的操作时间表，然后写出每一个微操作命令(控制信号)的逻辑表达式，最后根据逻辑表达式画出相应的组合逻辑电路图。

(1) 列出微操作命令的操作时间表

表6-1列出了上述各条机器指令的微操作控制信号的操作时间表。表中FE、IND和EX为CPU工作周期标志，$T_0 \sim T_2$为节拍，I为间址标志，在取指周期的T_2时刻，若测得I=1，则置“1”IND触发器，进入间址周期；若I=0，则置“1”EX触发器，进入执行周期。同理，在间址周期的T_2时刻，若测得IND=0(表示一次间址)，则置“1”EX，进入执行周期；若测得IND=1(表示多次间址)，则继续间接寻址。在执行周期的T_2时刻如果没有中断请求，则置“1”FE，进入下一条指令的取指周期。为简单起见，表中空格中“0”默认未画出。

(2) 进行微操作信号综合

在列出微操作时间表之后，即可对它们进行综合分析、归类，根据微操作时间表可以写出各微操作控制信号的逻辑表达式。表达式一般包括下列因素：

$$微操作控制信号=机器周期 \wedge 节拍 \wedge 脉冲 \wedge 操作码 \wedge 机器状态条件$$

表 6-1　微操作时间表

周期标志	节拍	状态条件	微操作信号	CLA	COM	INC	SHR	CSL	STP	ADD	SUB	AND	LDA	STA	JMP	JZ	JC	JN
FE（取指）	T_0		(PC)→MAR	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
			Read	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	T_1		M(MAR)→MDR	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
			(PC)+1→PC	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	T_2		(MDR)→IR	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
			OP(IR)→ID	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
		I	1→IND							1	1	1	1	1	1	1	1	1
		$\bar{I}$	1→EX	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
IND（间址）	T_0		(MDR)→MAR							1	1	1	1	1	1	1	1	1
			Read							1	1	1	1	1	1	1	1	1
	T_1		M(MAR)→MDR							1	1	1	1	1	1	1	1	1
	T_2		(MDR)→MAR							1	1	1	1	1	1	1	1	1
		$\overline{IND}$	1→EX							1	1	1	1	1	1	1	1	1
EX（执行）	T_0		(MDR)→MAR							1	1	1	1	1				
			Read							1	1	1	1					
	T_1		(AC)→MDR											1				
			Write											1				
			M(MAR)→MDR							1	1	1	1					
	T_2		(AC)+(MDR)→AC							1								
			(AC)−(MDR)→AC								1							
			(AC)∧(MDR)→AC									1						
			(MDR)→M(MAR)											1				
			(MDR)→AC										1					
			0→AC	1														
			$(\overline{AC})$→AC		1													
			(AC)+1→AC			1												
			R(AC)→AC AC_0→AC_0				1											
			L(AC)→AC AC_0→AC_n					1										
			Ad(MDR)→PC												1			
		Z	Ad(MDR)→PC													1		
		C	Ad(MDR)→PC														1	
		N	Ad(MDR)→PC															1
			0→G						1									

例如,根据微操作时间表写出 M(MAR)→MDR 逻辑表达式,并进行适当的简化:

$$
\begin{aligned}
&M(MAR)\to MDR\\
&=FE\cdot T_1+IND\cdot T_1(ADD+SUB+AND+LDA+STA+JMP+JZ+JC+JN)\\
&\quad+EX\cdot T_1(ADD+SUB+AND+LDA)\\
&=T_1\{FE+IND(ADD+SUB+AND+LDA+STA+JMP+JZ+JC+JN)\\
&\quad+EX(ADD+SUB+AND+LDA)\}
\end{aligned}
$$

式中 ADD、SUB 等均来自操作码译码器的输出。

(3) 画出微操作命令的逻辑图

根据逻辑表达式可画出对应每一个微操作控制信号的逻辑电路图,并用逻辑门电路实现之。

6.5.3 微程序控制单元设计

微程序设计控制单元的主要任务是编写对应各条机器指令的微程序,具体步骤是首先写出对应机器指令的全部微操作节拍安排,然后确定微指令格式,最后编写出每条微指令的二进制代码。

1. 微程序控制单元的设计步骤

(1) 确定微程序控制方式

根据计算机系统的性能指标(主要是速度)确定微程序控制方式。如是采用水平微程序设计还是采用垂直微程序设计,微指令是按串行方式运行还是按并行方式运行等。

(2) 拟定微命令系统

初步拟定微命令系统,并同时进行微指令格式的设计,包括微指令字段的划分、编码方式的选择、初始微地址和后继微地址的形成等。

(3) 编制微程序

对微命令系统、微指令格式进行反复的核对和审查,并进行适当的修改;对重复和多余的微指令进行合并和精简,直至编制出全部机器指令的微程序为止。

(4) 微程序代码化

将修改完善的微程序转换成二进制代码,这一过程称为代码化或代真。代真工作可以用人工实现,也可以在机器上用程序实现。

(5) 写入控制存储器

最后将一串串二进制代码按地址写入控制存储器的对应单元。

2. 设计举例

为了便于与组合逻辑设计比较,仍以前述的15条机器指令为例,而且假设CPU结构与组合逻辑设计相同。为简化起见,不考虑间接寻址的情况。

由于微命令的数目不多,故采用直接控制方式,即微指令控制字段的每一位直接控制一个微操作。微程序的后继微地址的形成方法采用增量方式,在微指令中不设顺序控制字段。

每执行一条微指令，μMAR 自动加 1。

取指微程序的入口地址是控存的 00H 单元，机器启动后，μMAR 自动指向 00H 单元。取指微程序从主存中取出一条机器指令送入 IR，再根据机器指令的操作码变换成相应的微程序入口地址，实现一级功能转移。

当一条机器指令执行完毕后，应当转去执行下一条机器指令，即使该机器指令对应的微程序的最后一条微指令执行完后转向取指微程序。为了简化设计，在微指令中专门设置了一个机器指令执行完的标志。每一条机器指令的最后一条微指令中令该位为“1”，当执行到这条微指令时，使 μMAR 清 0，指向控存中取指微程序的入口地址，下一条要执行的就是取指微指令了。

本系统总共需要 25 个微命令，其中：

第 0 位 (PC)→MAR

第 1 位 Read

第 2 位 M(MAR)→MDR

第 3 位 (PC)+1→PC

第 4 位 (MDR)→IR

第 5 位 0→AC

第 6 位 $(\overline{AC})$→AC

第 7 位 (AC)+1→AC

第 8 位 R(AC)→AC，AC_0→AC_0

第 9 位 L(AC)→AC，AC_0→AC_n

第 10 位 0→G

第 11 位 (MDR)→MAR

第 12 位 (AC)+(MDR)→AC

第 13 位 (AC)−(MDR)→AC

第 14 位 (AC)∧(MDR)→AC

第 15 位 (MDR)→AC

第 16 位 (AC)→MDR

第 17 位 Write

第 18 位 MDR→M(MAR)

第 19 位 (MDR)→PC

第 20 位 $Z\cdot(MDR)+\overline{Z}\cdot(PC)\to PC$

第 21 位 $C\cdot(MDR)+\overline{C}\cdot(PC)\to PC$

第 22 位 $N\cdot(MDR)+\overline{N}\cdot(PC)\to PC$

第 23 位 微指令转移标志，该位=0，转取指微程序的入口地址(0 号单元)或转各机器指令微程序的入口地址(操作码字段后加 11 得到)；该位=1，微指令顺序执行。

第 24 位 一条机器指令执行完标志，该位=1，表示该指令执行完。

表 6-2 列出了对应 15 条机器指令的微指令码点。

表 6-2　微指令码点

微程序名称	微指令地址	微指令(二进制代码)																								
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
取指	00H	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
	01H	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
	02H	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
CLA	03H	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
COM	07H	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
INC	0BH	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
SHR	0FH	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
CSL	13H	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
STP	17H	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
ADD	1BH	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0
	1CH	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
	1DH	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1
SUB	1FH	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0
	20H	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
	21H	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1
AND	23H	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0
	24H	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
	25H	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1
LDA	27H	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0
	28H	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
	29H	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1
STA	2BH	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0
	2CH	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	1	0
	2DH	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1
JMP	2FH	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1
JZ	33H	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1
JC	37H	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1
JN	3BH	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1

6.6　流水线技术

对于指令的执行,可有几种控制方式:顺序方式、重叠方式、先行控制及流水线控制方式。顺序方式指的是各条机器指令之间顺序串行的执行,即执行完一条指令后,方可取出下一条指令来执行。这种方式控制简单,但速度慢,机器各部件的利用率低。为了加快指令的执行速度,充分利用计算机系统的硬件资源,提高机器的吞吐率,计算机中常采用重叠方式、先行控制方式以及流水线控制方式。

6.6.1 重叠控制

通常,一条指令的运行过程可以分为3个阶段:取指、分析、执行。假定每个阶段所需的时间为t,那么在无重叠(顺序)的情况下,需要$3t$才能得到一条指令的执行结果,如图6-21(a)所示。故采用顺序方式执行n条指令所需的时间为:

$$T = 3nt$$

如果每个阶段所需时间各为$t_{取指}$、$t_{分析}$和$t_{执行}$,则顺序执行n条指令所需时间为:

$$T = \sum(t_{取指} + t_{分析} + t_{执行})$$

最早出现的重叠是“取指$K+1$”和“执行K”在时间上的重叠,称为一次重叠,如图6-21(b)所示,这将使处理机速度有所提高,所需执行时间减少为:

$$T = 3\times t + (n-1)\times 2t = (2\times n + 1)t$$

一次重叠方式需要增加一个指令缓冲器,在执行第K条指令时,寄存所取出的第$K+1$条指令。如果进一步增加重叠,使“取指$K+2$”“分析$K+1$”和“执行K”重叠起来,称为二次重叠(见图6-21(c)),则处理机速度还可以进一步提高,所需执行时间减少为:

$$T = 3\times t + (n-1)t = (2+n)t$$

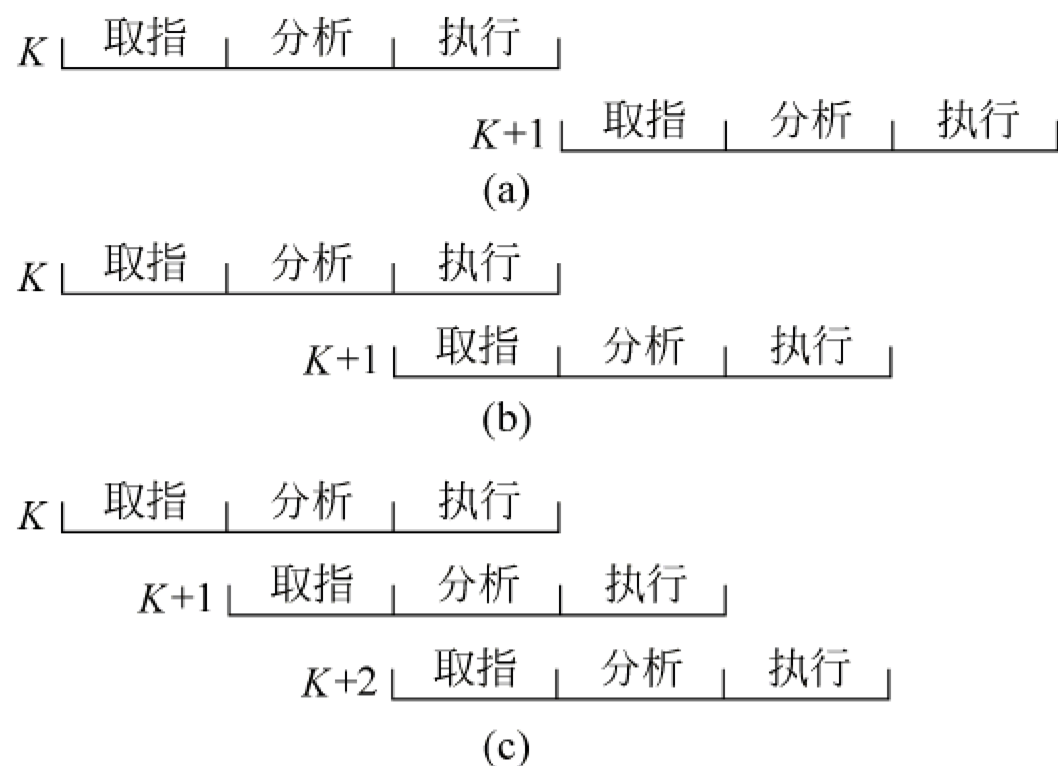


图6-21 重叠控制方式

为了能在“执行K”的同时,完成“分析$K+1$”和“取指$K+2$”的工作,就需要控制器同时发出3个阶段所需的控制信号。为此,应把CPU中原来集中的控制器,分解为存储控制器、指令控制器和运算控制器。

如果在“分析$K+1$”时需要访存取出操作数,而“取指$K+2$”时也需访存取指令,此时就会出现访存冲突。为了解决这个问题,第一种方法是设置两个存储器,分别用来存放操作数和指令,即采用哈佛结构。第二种方法是主存采用多体交叉存储结构,指令和操作数仍混存于主存中,只要第$K+1$条指令的操作数和第$K+2$条指令本身不在同一存储体内,就能在一个存储周期内同时取出两者。第三种方法是设置指令缓冲器(指令预取队列),预先将未执行到的下一条指令由主存中取到指令缓冲器去,这样,“取指$K+2$”时只需将第$K+2$条指令由指令缓冲器中拿出来送到指令寄存器去,而无须访问主存了。

很明显,指令的重叠执行并不能加快单条指令的执行时间,但可以加快相邻两条、多条指令乃至整个程序段的执行时间。

指令的重叠执行对于大多数非分支程序来说可以提高执行速度;但如果遇到转移、转子指令和各种中断,或者遇到第 K 条指令的执行结果正巧是第 $K+1$ 条指令的操作数的情况(数据相关)时,提前取出的指令将是无效的,此时重叠也就失败了。

6.6.2 先行控制原理

假设每次都可以在指令缓冲器中取得指令,则取指阶段就可合并到分析阶段中,指令的运行过程就变为分析和执行两个阶段了。如果所有指令的"分析"与"执行"的时间均相等,则重叠的流程是非常流畅的,机器的指令分析部件和执行部件功能充分地发挥,机器的速度也能显著地提高。但是,现代计算机的指令系统很复杂,各种类型指令难于做到"分析"与"执行"时间始终相等,此时,各个阶段的控制部件就有可能出现间断等待的问题。在图 6-22 中,分析部件在"分析 $K+1$"和"分析 $K+2$"之间有一个等待时间 Δt_1,在"分析 $K+2$"和"分析 $K+3$"之间又有一个等待时间 Δt_2;执行部件在"执行 $K+2$"和"执行 $K+3$"之间有一个等待时间 Δt_3。指令的分析部件和执行部件都不能连续地、流畅地工作,从而使机器的整体速度受到影响。

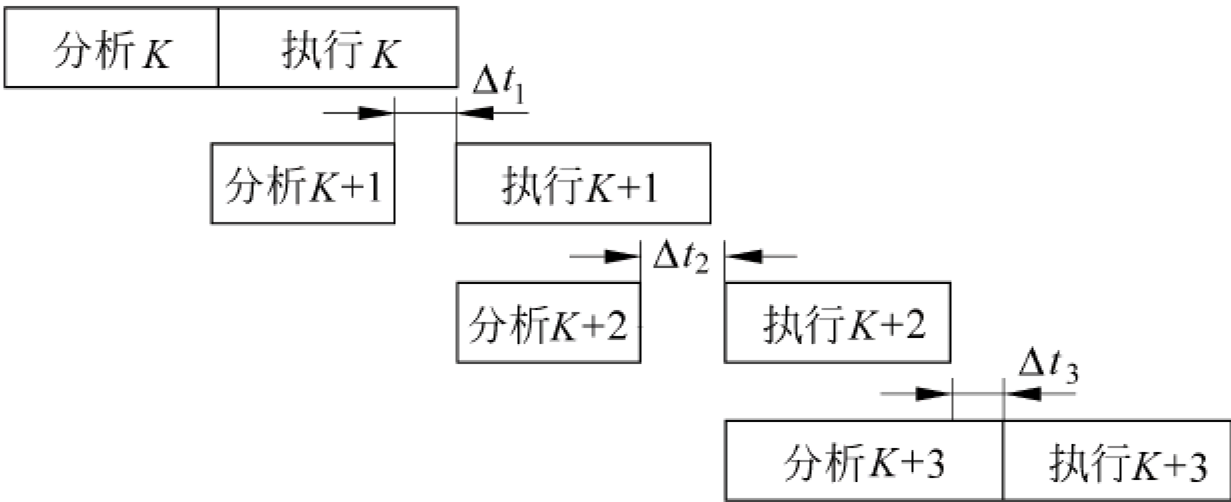


图 6-22 "分析"和"执行"时间不等的重叠

由于分析和执行部件有时处于空闲状态,此时执行 n 条指令所需时间为:

$$T = t_{分析1} + \sum_{i=2}^{n}[\max\{t_{分析i}, t_{执行i-1}\}] + t_{执行n}$$

为了使各部件能连续地工作,提出了先行控制的方式,如图 6-23 所示。虽然图中"分析"和"执行"阶段之间有等待的时间间隔 Δt_i,但它们各自的流程中却是连续的。先行控制的主要目的是使各阶段的专用控制部件不间断的工作,以提高设备的利用率及执行速度。

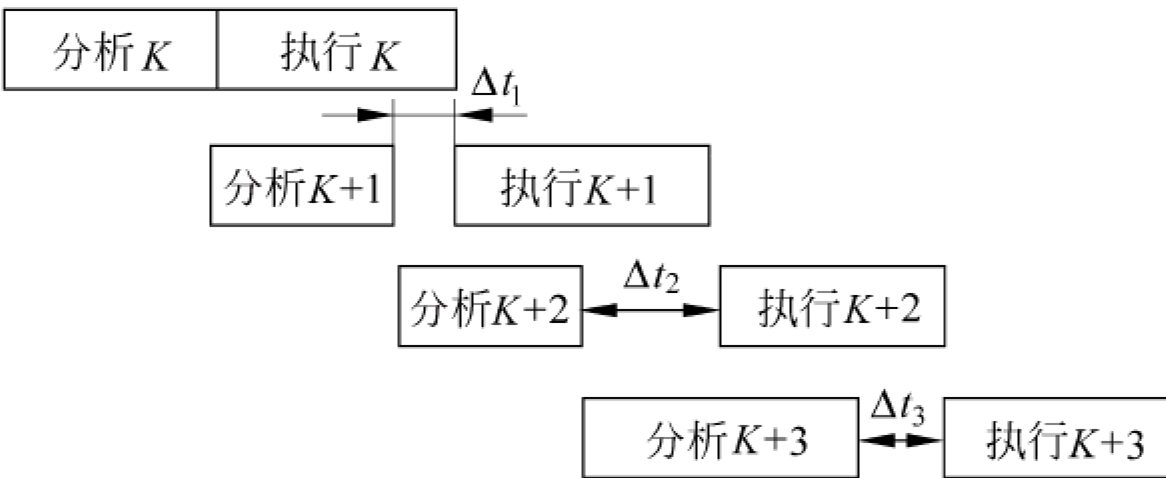


图 6-23 先行控制方式的时序

由于分析和执行部件能分别连续不断地分析和执行指令,此时执行 n 条指令所需时间为:

$$T_{先行}=t_{分析1}+\sum_{i=1}^{n}t_{执行i}$$

6.6.3 流水工作原理

流水处理技术是在重叠、先行控制方式的基础上发展起来的，它基于重叠的原理，但却是在更高程度上的重叠。

1. 流水线

流水线是将一个较复杂的处理过程分成 m 个复杂程度相当、处理时间大致相等的子过程，每个子过程由一个独立的功能部件来完成，处理对象在各子过程连成的线路上连续流动。在同一时间，m 个部件同时进行不同的操作，完成对不同对象的处理。这种方式类似于现代工厂的生产流水线，在那里每隔一段时间（Δt）从流水线上流出一个产品，而生产这个产品的总时间要比 Δt 大得多。由于流水线上各部件并行工作，机器的吞吐率将大大提高。例如，将一条指令的执行过程分成取指令、指令译码、取操作数和执行 4 个子过程，分别由 4 个功能部件来完成，每个子过程所需时间为 Δt，4 个子过程的流水线如图 6-24(a)所示。

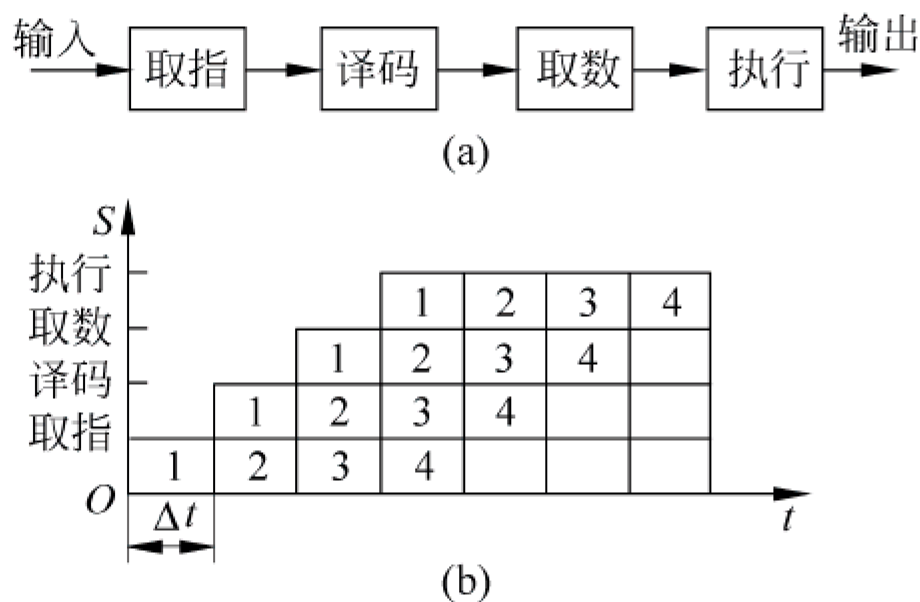


图 6-24　4 个子过程的流水处理

图 6-24(b)是流水线工作的时空图。图中横坐标为时间，纵坐标为空间（即各子过程），标有数字的方格说明占用该空间与时间的任务号，在本例中表示机器处理的第一、二、三、四条指令，最多可以有 4 条指令在不同的部件中同时进行处理。若执行一条指令所需时间为 T，那么在理想情况下，当流水线充满后，每隔 $\Delta t=\frac{T}{4}$ 就完成了一条指令的执行。图中子过程数 $m=4$，任务数 $n=4$。

2. 流水线分类

按照不同角度，流水线可有多种不同分类方法。

（1）按处理级别分类

流水线按处理级别可分为操作部件级、指令级和处理机级 3 种。操作部件级流水线是将复杂的算术逻辑运算组成流水线工作方式。例如，可将浮点加法操作分成求阶差、对阶、尾数相加以及结果规格化 4 个子过程。指令级流水线则是将指令的整个执行过程分成多个子过程，如前面提到的取指令、指令译码、取操作数和执行 4 个子过程。处理机级流水线又称宏流水线，如图 6-25 所示，这种流水线由两个或两个以上处理机通过存储器串行连接起来，每个处理机对同一数据流的不同部分分别进行处理。各个处理机的输出结果存放在与下一个处理机所共享的存储器中。每个处理机完成某一专门任务。

（2）按功能分类

流水线按功能可分成单功能流水线和多功能流水线两种。单功能流水线只能实现一种

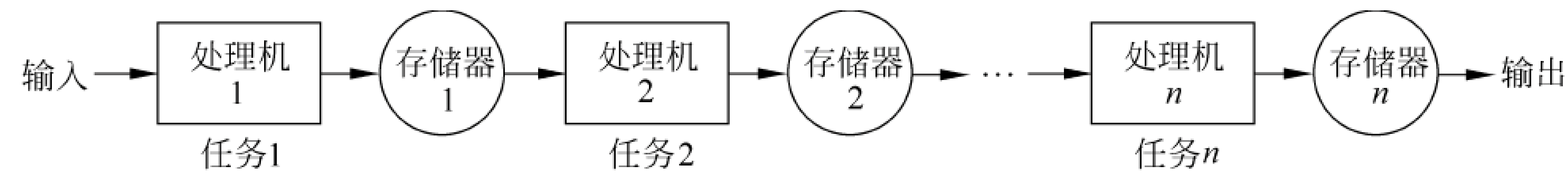


图 6-25 处理机级流水线

固定的功能,例如,浮点加法流水线专门完成浮点加法运算,浮点乘法流水线专门完成浮点乘法运算。多功能流水线则可有多种连接方式来实现多种功能,例如,美国 TI 公司生产的 ASC 计算机中的一个多功能流水线,共有 8 个功能段(见图 6-26(a)),按需要它可将不同的功能段连接起来完成某一功能,以实现定点加法(见图 6-26(b))、浮点加法(见图 6-26(c))和定点乘法(见图 6-26(d))等功能。

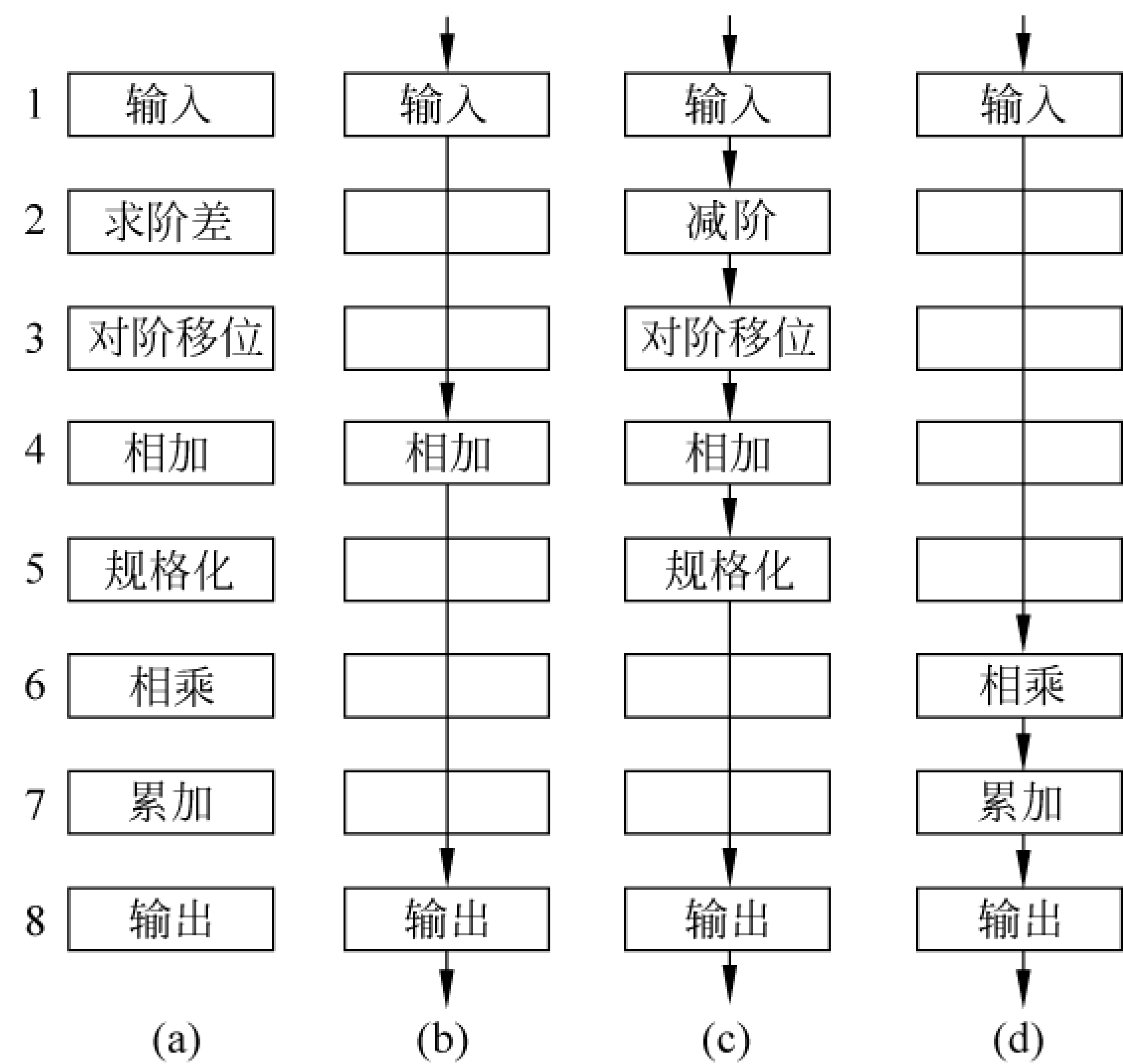


图 6-26 TI-ASC 计算机的多功能流水线

(3) 按工作方式分类

多功能流水线按工作方式可分为静态流水线和动态流水线两种。

静态流水线在同一时间内各段只能以一种功能连接流水,当从一种功能连接变为另一种功能连接时,必须先排空流水线,然后为另一种功能设置初始条件后方可使用。显然,不希望这种功能的转换频繁发生,否则将严重影响流水线的处理效率。

动态流水线则允许在同一时间内将不同的功能段连接成不同的功能子集,以完成不同的功能。

(4) 按流水线结构分类

流水线按结构分为线性流水线和非线性流水线两种。在线性流水线中,从输入到输出,每个功能段只允许经过一次,不存在反馈回路。一般的流水线均属这一类。非线性流水线除有串行连接通路外,还有反馈回路,在流水过程中,某些功能段要反复多次使用。

3. 指令流水线的相关性

对于指令流水线,相邻或相近的两条指令可能会因为存在某种关联,后一条指令不能按

照原指定的时钟周期运行,使流水线断流。指令流水线的相关性包括结构相关、数据相关、控制相关。

(1) 结构相关

由于多条指令在同一时刻争夺同一资源而形成的冲突称为结构相关,也称资源相关。

(2) 数据相关

后续指令要使用前面指令的操作结果,而这一结果尚未产生或者未送到指定的位置,从而造成后续指令无法运行的局面称为数据相关。

根据指令间对同一个寄存器读或写操作的先后次序关系,数据相关可分为 RAW(写后读)、WAR(读后写)和 WAW(写后写)3 种类型。例如,有 i 和 j 两条指令,i 指令在前,j 指令在后,则 3 种不同类型的数据相关的含义如下。

RAW:指令 j 试图在指令 i 写入寄存器前就读出该寄存器内容,这样指令 j 就会错误地读出该寄存器旧的内容。

WAR:指令 j 试图在指令 i 读出该寄存器前就写入该寄存器,这样指令 i 就会错误地读出该寄存器的新内容。

WAW:指令 j 试图在指令 i 写入寄存器前就写入该寄存器,这样两次写的先后次序被颠倒,就会错误地使由指令 i 写入的值成为该寄存器的内容。

上述的 3 种数据相关,在按序流动的流水线中,只可能出现 RAW 相关;在非按序流动的流水线中,既可能发生 RAW 相关,也可能发生 WAR 和 WAW 相关。

(3) 控制相关

控制相关主要是由转移指令引起的,在遇到条件转移指令时,存在着是顺序执行还是转移执行两种可能,需要依据条件的判断结果来选择其一。在无法确定应该选择把哪一程序段安排在转移指令之后来执行的局面称为控制相关,又称指令相关。

6.7 精简指令系统计算机

精简指令系统计算机(RISC)是 20 世纪 80 年代提出的一种新的设计思想,目前运行中的许多计算机都采用了 RISC 体系结构或采用了 RISC 设计思想。

6.7.1 RISC 的特点和优势

1. RISC 的主要特点

目前,难以在 RISC 和 CISC 之间划出一条明显的分界线,但大部分 RISC 具有下列一些特点:

- 指令总数较少(一般不超过 100 条);
- 基本寻址方式种类少(一般限制在 2 或 3 种);
- 指令格式少(一般限制在 2 或 3 种),而且长度一致;
- 除取数和存数指令(LOAD/STORE)外,大部分指令在单周期内完成;
- 只有取数和存数指令能够访问存储器,其余指令的操作只限于在寄存器之间进行;
- CPU 中通用寄存器的数目应相当多(32 个以上,有的可达上千个);

- 为提高指令执行速度,绝大多数采用硬连线控制实现,不用或少用微程序控制实现;
- 采用优化的编译技术,力求以简单的方式支持高级语言。

表6-3列出了CISC和RISC的区别。

表6-3 CISC和RISC的区别

指令系统体系结构	CISC	RISC
指令系统	复杂,庞大	简单,精简
指令数目	一般大于200条	一般小于100条
指令字长	不固定	等长
寻址方式	一般大于4	一般小于4
可访存指令	不加限制	只有LOAD/STORE指令
各种指令执行时间	相差较大	绝大多数在一个周期内完成
通用寄存器数量	较少	多
控制方式	绝大多数为微程序控制	绝大多数为硬布线控制

2. RISC的优势

计算机执行一个程序所用的时间t可用下式表示:

$$t = I \times C \times T$$

式中,I是高级语言编译后在机器上执行的机器指令总数,C是执行每条机器指令所需的平均周期数,T是每个周期的执行时间。表6-4为RISC和CISC的统计数据,表中I、T为比值,C为实际周期数。

表6-4 RISC、CISC的I、C、T统计

指令集	I	C	T
RISC	1.2～1.4	1.3～1.7	<1
CISC	1	4～10	1

由于RISC机器的指令比较简单,故完成同样的任务要比CISC机器使用更多的指令,因此RISC的I要比CISC多20%～40%。但是,因为RISC的大多数指令只需单周期实现,所以C值要比CISC小得多。同时因为RISC结构简单,所以完成一个操作所经过的数据通路较短,使T值有所减少,根据上述统计折算下来,RISC的处理速度要比相同规模的CISC提高3～5倍。

由于RISC的结构简化,降低了芯片的复杂程度,节约了芯片面积。若使RISC芯片保持与CISC芯片相同的面积和复杂程度,则RISC芯片可集成更多的功能部件,集成度大为提高,且功能也大大加强。

当然,RISC也存在着某些局限性,因此实际上商品化的RISC机器并不是纯粹的RISC。为了满足应用的需要,实用的RISC除了保持RISC的基本特色之外,还必须辅以一些必不可少的复杂指令,如浮点和十进制运算指令等。所以,这种机器实际上是在RISC基础上实现了RISC与CISC的完美结合。

6.7.2 RISC 基本技术

为了能有效地支持高级语言并提高 CPU 的性能，RISC 结构采用了一些特殊技术。

1. RISC 寄存器管理技术

计算机中最慢的操作是访问存储器的操作，因此在 RISC 中，为了减少访存的频度，通常在 CPU 芯片上设置大量寄存器，把常用的数据保存在这些寄存器中。例如，RISC Ⅱ 有 138 个寄存器，AM 29000 有 192 个寄存器，Ry 公司的 9000 系列超级小型机中甚至设置了多达 528 个寄存器。

在 RISC Ⅱ 中使用了重叠寄存器窗口技术，即设置一个数量比较大的寄存器堆，并把它划分成很多窗口。每个过程使用其中相邻的 3 个窗口和一个公共的窗口，而在这相邻的 3 个窗口中有一个窗口与前一个过程公用，还有一个窗口是与下一个过程公用的。

2. 流水线技术

一条指令通常可分为取指、译码、执行、写回等多个阶段，要想在一个周期内串行完成这些操作是不可能的，因此，采用流水线技术势在必行。

流水线的基本概念已在前面介绍过，各种 RISC 采用的流水线结构不完全相同。例如，RISC Ⅰ 采用两级流水线（取指、执行），RISC Ⅱ 采用三级流水线（取指、执行、写回），Am 29000 则采用四级流水线（取指、译码、执行、写回）。

当出现数据相关和程序转移情况时，流水线结构就可能发生断流的问题，这将会影响流水线的效率。

对于两级流水线不存在数据相关问题，而流水线级数越多，情况越复杂。RISC Ⅱ 是采用内部推前的方法来解决数据相关的问题的。每当执行 LOAD/STORE 指令时，就把流水线各级操作暂停一个周期，以完成存储器读写，所有指令的读写运算结果总是先放在结果暂存器中。当硬件检测到数据相关时，直接从结果暂存器取得源操作数，即将与第 $i+1$ 条指令操作有关的第 i 条指令的数据预先推入一个暂存器中，所以第 $i+1$ 条指令是从暂存器中取出操作数的，这样使流水线不至于阻塞。

3. 延时转移技术

在流水线中，取下一条指令是同上一条指令的执行并行进行的，当遇到转移指令时，流水线就可能断流。RISC 机器中，当遇到转移指令时，可以采用延迟转移方法或优化延迟转移方法。在采取延迟转移方法时，编译程序自动在转移指令之后插入一条（或几条，根据流水线情况而定）空指令，以延迟后继指令进入流水线的时间。所谓优化延迟转移方法，是将转移指令与前条指令对换位置，提前执行转移指令，可以节省一个机器周期。

6.8 微处理器中的新技术

6.8.1 超标量和超流水线技术

在 RISC 之后，出现了一些提高指令级并行性的技术，使得计算机在每个时钟周期里可

以解释多条指令,这就是超标量技术和超流水线技术。

前面提到的流水线技术是指常规的标量流水线,每个时钟周期平均执行的指令的条数小于或等于1,即它的指令级并行度(Instruction Level Parallelism,ILP)$\leqslant 1$。

超标量技术是通过重复设置多个功能部件,并让这些功能部件同时工作来提高指令的执行速度,实际上是以增加硬件资源为代价来换取处理器性能的。使用超标量技术的处理器在一个时钟周期内可以同时发射多条指令,假设每个时钟周期发射 m 条指令,则有 $1<\mathrm{ILP}<m$。

超流水线仍然是一种流水线技术,可以认为它是将标量流水线的子过程(段)再进一步细分,使得子过程数(段数)大于或等于8的情况。也就是说,只需要增加少量硬件,通过各部分硬件的充分重叠工作来提高处理器性能。采用超流水线技术的处理器在一个时钟周期内可以分时发射多条指令,假设每个时钟周期 Δt 分时地发射 n 条指令,则每隔 $\Delta t'$ 就流出一条指令,此时 $\Delta t'=\Delta t/n$,有 $1<\mathrm{ILP}<n$。

6.8.2 EPIC 的指令级并行处理

EPIC 架构是 Itanium 挑战 RISC 架构的基础,它的设计思想就是用智能化的软件来指挥硬件,以实现指令级并行计算。采用 EPIC 架构的处理器在运行中,首先由编译器分析指令之间的依赖关系,将没有依赖关系的3条指令组合成一个128位的指令束。在低端 CPU 中,每个时钟周期调度1个指令束,CPU 等待所有的指令都执行完后再调度下一个指令束。在高端的 CPU 中,每个时钟周期可以调用多个指令束,类似于现在的超标量设计。另外,在高端 CPU 中,CPU 可以在原有的指令束没有执行完之前调度新的指令束。当然,它需要检查将要用到的寄存器和功能单元是否可用,但是它不用检查同一束中的其他指令是否和它冲突,因为编译器已经保证不会出现这种情况。

值得一提的是,EPIC 还采用了更为先进的分支判定技术来保证并行处理的稳定性。传统 CPU 采用的分支预测技术是只沿一个预测的分支执行,一旦预测错误就不得不清空整条流水线,从头再来,损失较大;EPIC 的分支判定技术则是同时执行两条分支,把条件分支指令变成可同时执行的判定指令,让两条分支并行执行,最后丢掉不需要的结果即可。

另外,EPIC 还导入了数据推测装载技术,它可预选在 Cache 中装入接下来的指令可能调用的数据,来提升 Cache 的工作效率,对经常需要使用 Cache 的应用程序(如大型数据库)的性能提升非常显著。

6.8.3 超线程技术

超线程(Hyper-Threading,HT)是 Intel 公司提出的一种提高 CPU 性能的技术,简单地说就是将一个物理 CPU 当作两个逻辑 CPU 使用,使 CPU 可以同时执行多重线程,从而发挥更大的效率。超线程技术利用特殊的硬件指令,把两个逻辑内核模拟成两个物理芯片,让单个处理器都能使用线程级并行计算,进而兼容多线程操作系统和应用软件,减少了 CPU 的闲置时间,提高了 CPU 的运行效率。

超线程技术可以使操作系统或者应用软件的多个线程同时运行于一个超线程处理器上,其内部的两个逻辑处理器共享一组处理器执行单元,并行完成加、乘、加载等操作。这样做可以使得处理器的处理能力提高30%,因为在同一时间里应用程序可以充分使用芯片的各个运算单元。

对于单线程芯片来说，虽然也可以每秒钟处理成千上万条指令，但是在某一时刻它只能够对一条指令(单个线程)进行处理，结果必然使处理器内部的其他处理单元闲置。而超线程技术则可以使处理器在某一时刻同步并行处理更多指令和数据(多个线程)。所以，超线程是一种可以将 CPU 内部暂时闲置处理资源充分"调动"起来的技术。

在处理多个线程的过程中，多线程处理器内部的每个逻辑处理器均可以单独对中断做出响应，当第一个逻辑处理器跟踪一个软件线程时，第二个逻辑处理器也开始对另外一个软件线程进行跟踪和处理了。另外，为了避免 CPU 处理资源冲突，负责处理第二个线程的那个逻辑处理器，其使用的仅是运行第一个线程时被暂时闲置的处理单元。例如，当一个逻辑处理器在执行浮点运算(使用处理器的浮点运算单元)时，另一个逻辑处理器则可以执行加法运算(使用处理器的整数运算单元)。这样做，无疑大大提高了处理器内部处理单元的利用率和相应数据、指令的吞吐能力。

超线程技术实现的前提条件是需要五大支持，即 CPU 支持、主板芯片组支持、主板 BIOS 支持、操作系统支持和应用软件支持。只有满足这些条件，才能使得系统效能得到提升。需要指出的是，超线程技术仅在多任务处理时有优势，在进行单个任务处理时，优势表现不出来，而且因为打开了超线程(在 BIOS 中)，处理器内部缓存就会被划分成几个区域，互相共享内部资源，反而会造成单个的子系统性能下降。

超线程技术最早出现在 2002 年推出的 Pentium 4 上，但由于当时支持超线程技术的应用软件缺少，使得它的优势无法发挥，之后 Intel 公司推出的一些处理器也曾不再使用这个技术。直至 2009 年 Core i 系列的诞生，超线程技术才有卷土重来之势，如四核的 Core i7 可同时支持 8 个线程工作，大幅增强其多线程性能。

6.8.4 双核与多核技术

1. 双核处理器

双核处理器是指在一个处理器上集成两个运算核心，从而提高计算能力。"双核"的概念最早是由 IBM、HP、Sun 等支持 RISC 架构的高端服务器厂商提出的，目前双核处理器已在微机中普遍使用，图 6-27 给出了 Intel 双核的基本结构。

双核处理器并不能达到 1+1=2 的效果，也就是说，双核处理器并不会比同频率的单核处理器提高一倍的性能。IBM 公司曾经对比了 AMD 双核处理器和单核处理器的性能，其结果是双核和单核相比大概性能提高 60%。不过值得一提的是，这个 60%并不是说处理同一个程序时的提升幅度，而是在多线程任务下得到的提升。换句话说，双核处理器的优势在于多线程应用，如果只是处理单个任务，运行单个程序，也许双核处理器与同频率的单核得到的效果是一样的。

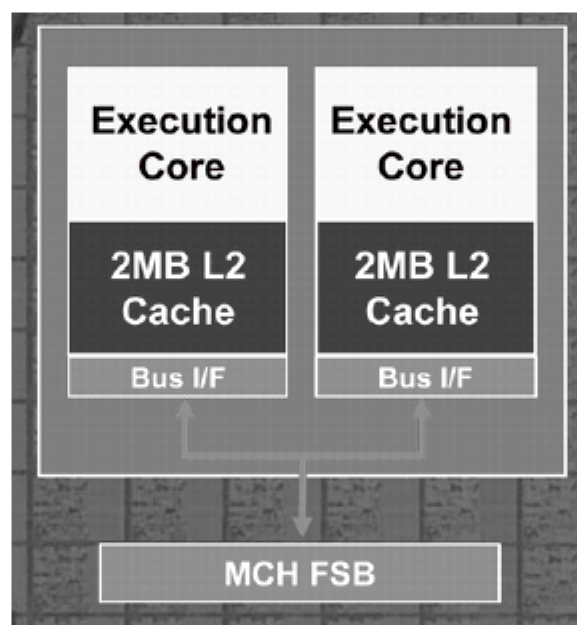


图 6-27　Intel 双核的基本结构

2. 超线程技术与双核心技术的区别

开启了超线程技术的 Pentium 4(单核)与 Pentium D(双核)在操作系统中都同样被识别为两个处理器，它们究竟是不是一样的呢？这个问题确实具有迷惑性。其实，可以简单地

把双核心技术理解为两个“物理”处理器,是一种“硬”的方式;而超线程技术只是两个“逻辑”处理器,是一种“软”的方式。

支持超线程的 Pentium 4 能同时执行两个线程,但超线程中的两个逻辑处理器并没有独立的执行单元、整数单元、寄存器甚至缓存等资源。它们在运行过程中仍需要共用执行单元、缓存和系统总线接口。在执行多线程时两个逻辑处理器均是交替工作,如果两个线程都同时需要某一个资源时,其中一个要暂停并要让出资源,要待那些资源闲置时才能继续。因此,超线程技术仅可以视为对单个处理器运算资源的优化利用。

而双核心技术则是通过“硬”的物理核心实现多线程工作的:每个核心拥有独立的指令集、执行单元,与超线程中所采用的模拟共享机制完全不一样。在操作系统看来,它是实实在在的双处理器,可以同时执行多项任务,能让处理器资源真正实现并行处理模式,其效率和性能提升要比超线程技术高得多,不可同日而语。

3. 多核多线程技术

目前,高性能微处理器研究的前沿逐渐从开发指令级并行(ILP)转向开发多线程并行(Thread Level Parallelism,TLP),单芯片多处理器(Chip Multiprocessor,CMP)就是实现TLP 的一种新型体系结构。

CMP 在一个芯片上集成多个微处理器核,每个微处理器核实质上都是一个相对简单的单线程微处理器或者比较简单的多线程微处理器,这样多个微处理器核就可以并行地执行程序代码,因而具有较高的线程级并行性。

如果按照单芯片多处理器上的处理器是否相同划分,可以分为同构 CMP 和异构 CMP,同构 CMP 大多数由通用的处理器组成,多个处理器执行相同或者类似的任务。异构 CMP 除含有通用处理器作为控制、通用计算之外,多集成了 DSP、ASIC、媒体处理器、VLIW 处理器等针对特定的应用提高计算的性能。

Pentium 系列微处理器中的 Pentium 属于单核单线程处理器,Pentium 4 属于单核多线程处理器,Pentium D 属于多核单线程处理器,Pentium EE 属于多核多线程处理器,这几种微处理器的内部结构示意图如图 6-28 所示,图中 EU 表示执行单元,CU 表示控制单元。

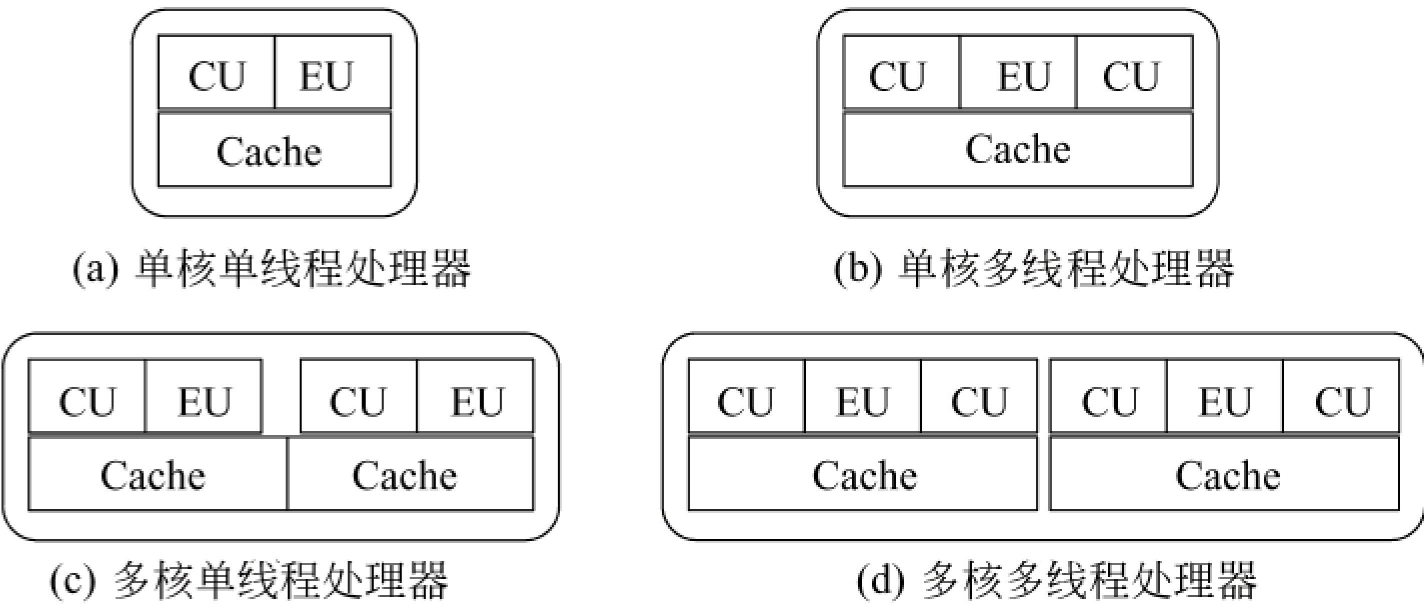


图 6-28 几种微处理器的内部结构

目前,市面上已大量出现三核、四核、六核和八核的处理器,甚至有十核和十二核的面向服务器和工作站的处理器,如 Intel 的 Xeon E5 就有八核十六进程。

多核处理器广泛受到青睐的一个主要原因是,当工作频率受限于技术进步时,并行处理

技术可以采用更多的内核并行运行来大大提高处理器的等效运行速度，同时由于工作频率没有提高，功耗相对于同性能的高频单核处理器要低得多。不难看出，多核处理器是处理器发展的必然趋势。无论是移动与嵌入式应用、桌面应用还是服务器应用，都将采用多核的架构。未来的多核处理器芯片将包含很多通用的处理器核，每个处理器核运行2～4个线程。同时芯片中包含成千个异构可编程加速器，用于媒体加速等特殊处理。

6.8.5 睿频加速技术

睿频加速技术的英文全称是 Intel Turbo Boost Technology，它是 Intel Core i5/i7 处理器的独有特性，也是 Intel 公司新宣布的一项技术。睿频加速技术可以理解为自动超频，实际上是一个新一代的能效管理方案。

当启动一个运行程序后，处理器会自动加速到合适的频率，而原来的运行速度会提升10%～20%，以保证程序流畅运行。应对复杂应用时，处理器可自动提高运行主频以提高速度，轻松进行对性能要求更高的多任务处理；当进行工作任务切换时，如果只有内存和硬盘在进行主要的工作，处理器会立刻处于节电状态，这样既保证了能源的有效利用，又使程序速度大幅提升。通过智能化地加快处理器速度，从而根据应用需求最大限度地提升性能，为高负载任务提升运行主频高达20%。

Turbo Boost，顾名思义就是加速技术，它是新一代 CPU 的趋势，使得 CPU 更加智能。CPU 会确定其当前工作功率、电流和温度是否已达到最高极限，如仍有多余空间，CPU 会逐渐提高活动内核的频率，以进一步提高当前任务的处理速度，当程序只用到其中的某些核心时，CPU 会自动关闭其他未使用的核心。当开启睿频加速之后，CPU 会根据当前的任务量自动调整 CPU 主频，从而是重任务时发挥最大的性能，是轻任务时发挥最大节能优势。睿频加速技术无须用户干预，自动实现，而且完全让 CPU 运行在技术规范内，安全可靠，不需要任何额外的投资，系统运行稳定，在不影响 CPU 的热功耗设计（TDP）情况下，能把核心工作频率调得更高。

Turbo Boost 是基于 CPU 的电源管理技术来实现的，通过分析当前 CPU 的负载情况，智能地完全关闭一些用不上的核心，把能源留给正在使用的核心，并使它们运行在更高的频率，从而提供更强的性能。而需要多个核心时，则动态开启相应的核心，智能调整频率。

在进入 Turbo 方式后，繁忙 CPU 内核的频率会提升一级，通常每个时钟提升步进是133MHz。以实际发布的一款 Core i7-870 为例，这是一款四核八线程的处理器，其 CPU 主频为2.93GHz。如果只有一个内核处于运行状态，这个内核可以提速至3.6GHz（最大睿频），相当于上5个台阶，增加了5×133MHz。如果只有2个内核处于运行状态，这2个内核可以提速至3.46GHz，相当于上了4个台阶，增加了4×133MHz。如果3个或者4个内核处于运行状态，这个处理器可以提速到3.2GHz，相当于上了2个台阶，增加了2×133MHz。

睿频加速2.0技术的英文全称是 Turbo Boost 2.0，相比睿频1.0来说，睿频2.0的特点有两点：一是更加智能、更高能效，睿频2.0不再受 TDP 限制，而是通过 CPU 内部温度进行监测，在 CPU 内部温度许可的情况下可以超过 TDP 提供更大的睿频幅度，不睿频时却更节能；二是 CPU 和 GPU 都可以睿频，而且可以一起睿频。简单地说，睿频加速技术2.0更智能、更高效。

2016 年,Intel 公司推出了面向发烧级平台的睿频加速 Max 技术 3.0(Turbo Boost Max Technology 3.0)。它可以将处理器的单线程、多核性能提升 15%以上。在睿频 3.0 技术中,将 1 或 2 颗体质最好的处理器核心定义为最佳性能核心,然后尽可能提升它们的频率,从而满足更复杂场景的应用需求。睿频加速 Max 技术 3.0 不会取代睿频加速技术 2.0。前者对后者进行了增强,可大幅提高最快内核的频率,从而让用户能够更加灵活地获得卓越的处理器性能。

习　　题

6-1　控制器有哪几种控制方式?各有何特点?

6-2　什么是三级时序系统?

6-3　控制器有哪些基本功能?它可分为哪几类?分类的依据是什么?

6-4　中央处理器有哪些功能?它由哪些基本部件组成?

6-5　中央处理器中有哪几个主要寄存器?试说明它们的结构和功能。

6-6　某计算机 CPU 芯片的主振频率为 8MHz,其时钟周期是多少微秒?若已知每个机器周期平均包含 4 个时钟周期,该机的平均指令执行速度为 0.8MIPS,试问:

(1) 平均指令周期是多少微秒?

(2) 平均每个指令周期含有多少个机器周期?

(3) 若改用时钟周期为 0.4μs 的 CPU 芯片,则计算机的平均指令执行速度又是多少 MIPS?

(4) 若要得到 40 万次/秒的指令执行速度,则应采用主频为多少 MHz 的 CPU 芯片?

6-7　以一条典型的单地址指令为例,简要说明下列部件在计算机的取指周期和执行周期中的作用。

(1) 程序计数器(PC)。

(2) 指令寄存器(IR)。

(3) 算术逻辑运算部件(ALU)。

(4) 存储器数据寄存器(MDR)。

(5) 存储器地址寄存器(MAR)。

6-8　什么是指令周期?什么是 CPU 周期?它们之间有什么关系?

6-9　指令和数据都存放在主存,如何识别从主存储器中取出的是指令还是数据?

6-10　CPU 中指令寄存器是否可以不要?指令译码器是否能直接对存储器数据寄存器 MDR 中的信息译码?为什么?以无条件转移指令 JMP A 为例说明。

6-11　设一地址指令格式如下:

@	OP	A

现在有 4 条一地址指令:LOAD(取数)、ISZ(加“1”为零跳)、DSZ(减“1”为零跳)、STORE(存数),在一台单总线单累加器结构的机器上运行,试排出这 4 个指令的微操作序列。

注意:*当排 ISZ 和 DSZ 指令时不要破坏累加寄存器 Acc 原来的内容。*

6-12　某计算机的 CPU 内部结构如图 6-29 所示。两组总线之间的所有数据传送通过 ALU。ALU 还具有完成以下功能的能力:

F=A;　　F=B

F=A+1;　　F=B+1

F=A−1;　　F=B−1

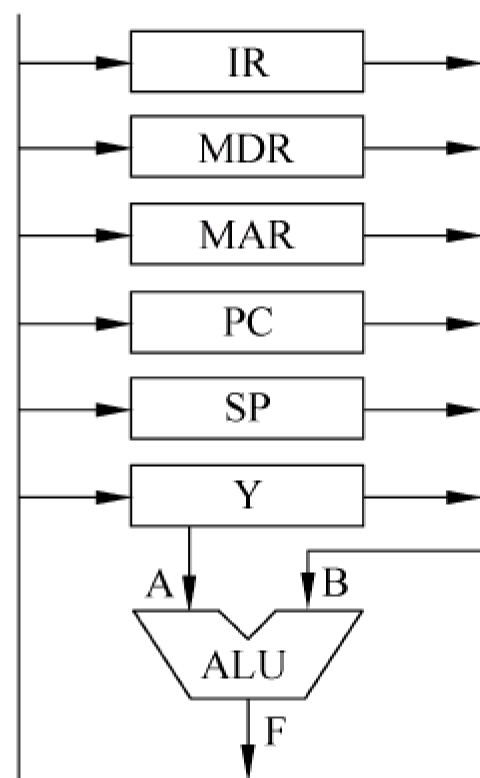


图 6-29　某计算机 CPU 内部结构

写出转子指令(JSR)的取指和执行周期的微操作序列。JSR 指令占两个字,第一个字是操作码,第二个字是子程序的入口地址。返回地址保存在存储器堆栈中,堆栈指示器始终指向栈顶。

6-13 某计算机主要部件如图 6-30 所示。

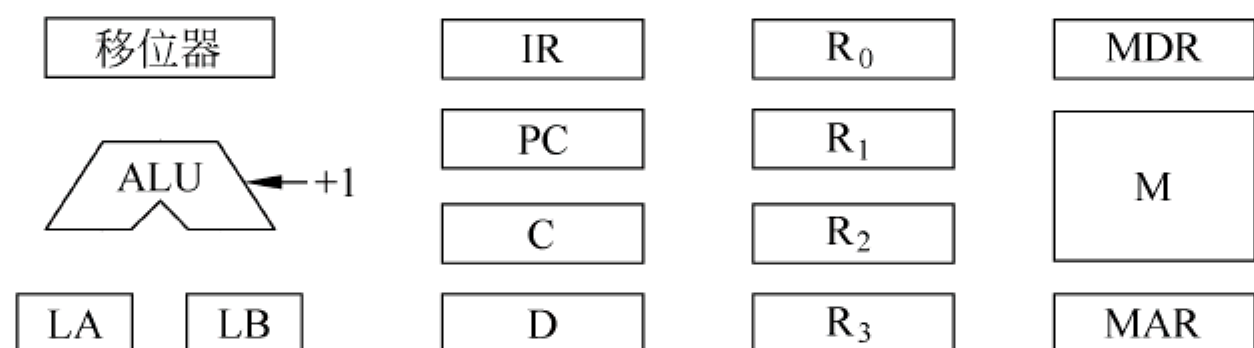


图 6-30 某计算机主要部件

LA——A 输入选择器 LB —— B 输入选择器 C、D —— 暂存器

(1) 补充各部件间的主要连接线,并注明数据流动方向。

(2) 写出指令 ADD (R_1),(R_2)+的执行过程(含取指过程与确定后继指令地址)。该指令的含义是进行加法操作,源操作数地址和目的操作数地址分别在寄存器 R_1 和 R_2 中,目的操作数寻址方式为自增型寄存器间址。

6-14 CPU 结构如图 6-31 所示,其中有一个累加寄存器 AC、一个程序状态字寄存器和其他 4 个寄存器,各部件之间的连线表示数据通路,箭头表示信息传送方向。

(1) 标明 4 个寄存器的名称。

(2) 简述指令从主存取出送到控制器的数据通路。

(3) 简述数据在运算器和主存之间进行存取访问的数据通路。

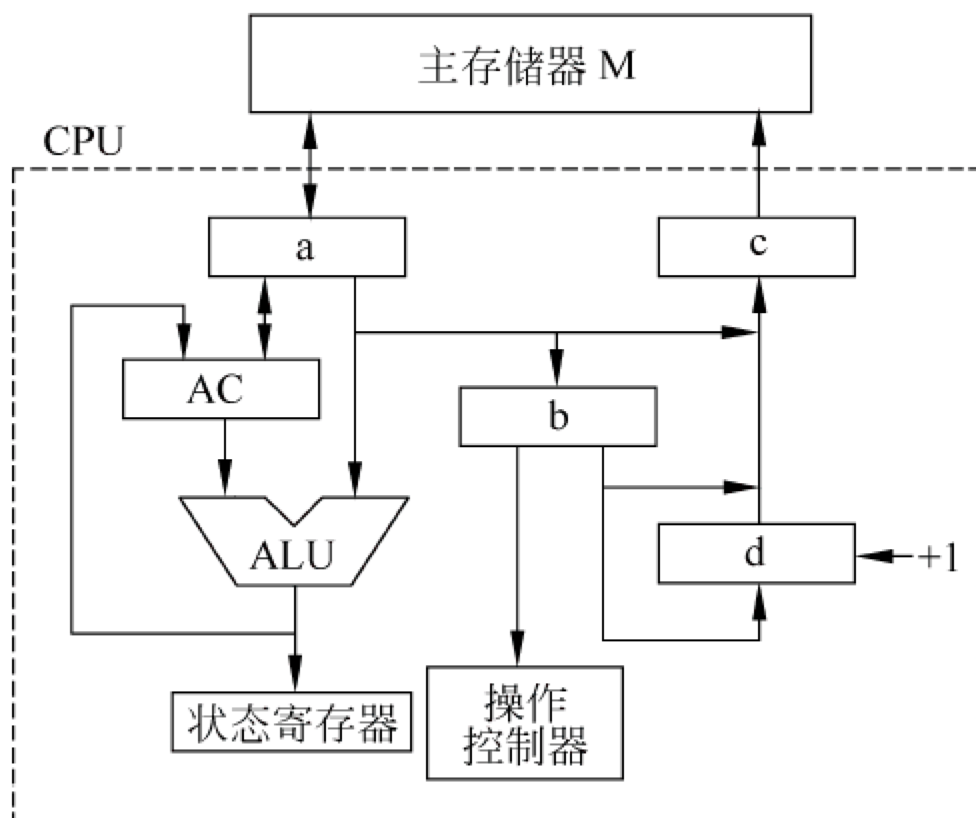


图 6-31 某计算机 CPU 结构

6-15 什么是微命令和微操作?什么是微指令?微程序和机器指令有何关系?微程序和程序之间有何关系?

6-16 什么是垂直型微指令?什么是水平型微指令?它们各有什么特点?又有什么区别?

6-17 水平型和垂直型微程序设计之间有什么区别?串行微程序设计和并行微程序设计有什么区别?

6-18 图 6-32 给出了某微程序控制计算机的部分微指令序列。图中每一框代表一条微指令。分支点 a 由指令寄存器(IR)的第 5 和 6 两位决定。分支点 b 由条件码 C_0 决定。已知微指令地址寄存器字长 8 位,现采用下址字段实现该序列的顺序控制。

(1) 设计实现该微指令序列的微指令字之顺序控制字段格式。

(2) 给出每条微指令的二进制编码地址。

(3) 画出微程序控制器的简化框图。

6-19 已知某计算机采用微程序控制方式,其控制存储器容量为 512×48 位,微程序可在整个控制存储器中实现转移,可控制转移的条件共 4 个,微指令采用水平型格式,后继指令地址采用断定方式,微指令

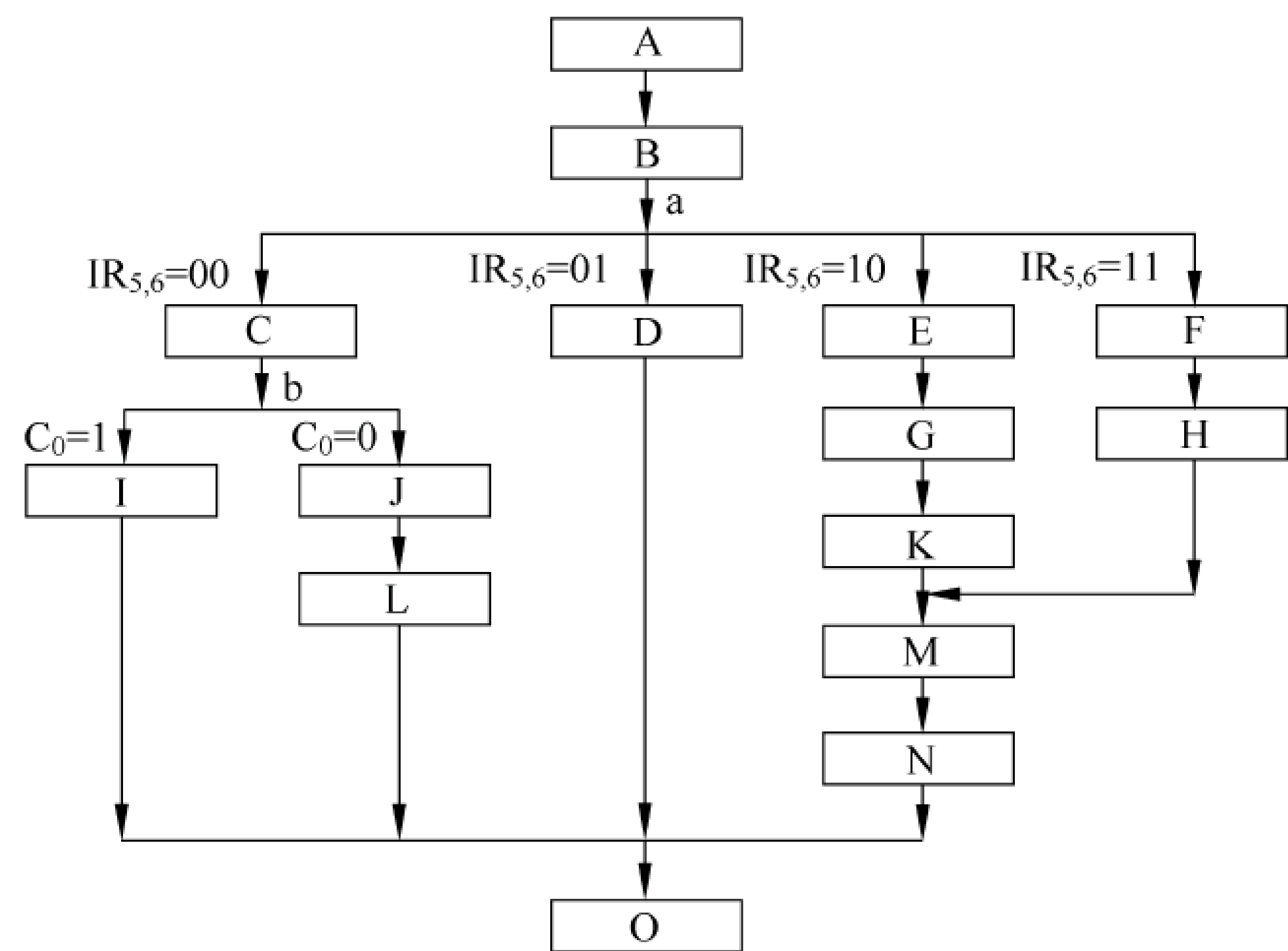


图 6-32　某计算机的部分微指令序列

格式如图 6-33 所示。

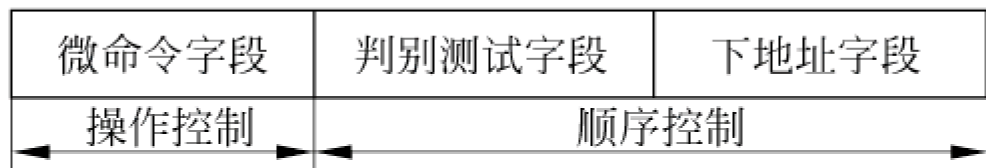


图 6-33　某计算机的微指令格式

(1) 微指令中的 3 个字段分别应为多少位？

(2) 画出围绕这种微指令格式的微程序控制器逻辑框图。

6-20　某计算机有 8 条微指令 $I_1 \sim I_8$，每条微指令所含的微命令控制信号如表 6-5 所示。

表 6-5　微指令所含微命令控制信号

微指令	微命令信号									
	a	b	c	d	e	f	g	h	i	j
I_1	√	√	√	√	√					
I_2	√			√		√	√			
I_3		√						√		
I_4			√							
I_5			√		√		√		√	
I_6	√							√		√
I_7			√	√				√		
I_8	√	√						√		

a～j 分别代表 10 种不同性质的微命令信号，假设一条微指令的操作控制字段为 8 位，安排微指令的操作控制字段格式，并将全部微指令代码化。

6-21　在微程序控制器中，微程序计数器(μPC)可以用具有加“1”功能的微地址寄存器(μMAR)来代替，试问程序计数器(PC)是否可以用具有加“1”功能的存储器地址寄存器(MAR)代替？

第 7 章 总线

在大多数计算机系统中，无论是计算机内部各部分之间，还是计算机与外部设备之间，数据传送都是通过总线(Bus)进行的。可以说，总线是计算机及其系统的重要组成部分。本章介绍总线的有关概念、总线仲裁方法、总线操作定时与常见总线标准。

7.1 总线概述

总线是一组能为多个部件分时共享的公共信息传送线路。共享是指总线上可以挂接多个部件，各个部件之间相互交换的信息都可以通过这组公共线路传送；分时是指同一时刻总线上只能传送一个部件发送的信息。

7.1.1 总线的基本概念

总线采用分时共享技术，当总线空闲(所有部件都以高阻状态连接在总线上)时，如果有一个部件要与目的部件通信，则发起通信的部件驱动总线，发出地址和数据。其他以高阻状态连接在总线上的部件，如果收到与自己相符的地址信息后，即接收总线上的数据。发送部件完成通信后，将总线让出(输出变为高阻态)。

1. 三态门和总线电路

高阻状态又称浮空状态，输出呈高阻状态即相当于输出开路(隔断)，输出端对地的电阻无限大，与外界断开联系。具备高阻状态的门电路称为三态门，即具有 3 种逻辑状态(逻辑 0、逻辑 1 和高阻状态)的门电路。三态门除了正常的输入端和输出端之外，还有一个控制端 G(或$\overline{G}$)。只有当控制端有效时，该三态门才能满足正常的逻辑关系；否则，输出将呈现高阻状态。根据输入输出关系和控制端的有效电平，可以分为 4 种类型的三态门，如图 7-1 所示。

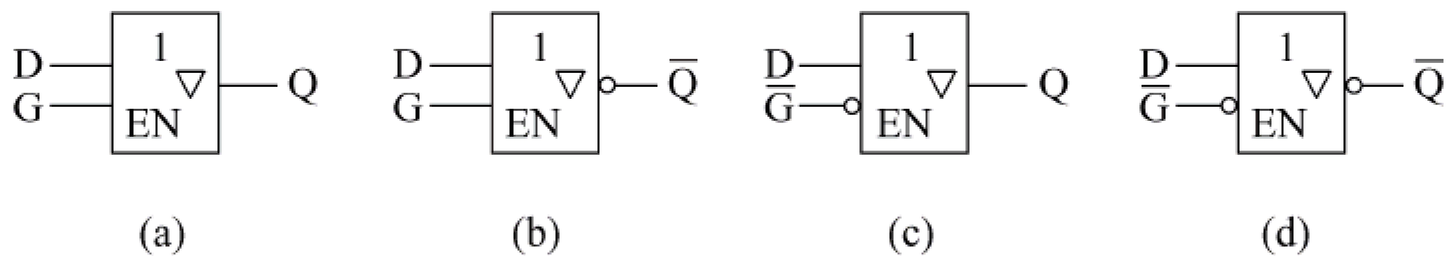


图 7-1　4 种类型的三态门

三态门主要用于总线连接,各个部件或设备必须通过三态缓冲器才能挂在总线上,通过控制端选择工作部件或设备。

按总线的逻辑结构来看,有单向总线和双向总线之分,单向总线是指总线上的信息只能向一个方向传送,双向总线是指总线上的信息可以向两个方向传送。例如,数据总线是双向三态的,既可以把CPU的数据传送到存储器或I/O设备,也可以将其他部件的数据传送到CPU;而地址总线是单向三态的,地址只能从CPU传向存储器或I/O设备。

2. 总线事务

通常把在总线上一对设备之间的一次信息交换过程称为一个"总线事务",把发出总线事务请求的部件称为主设备,与主设备进行信息交换的对象称为从设备。例如,CPU要求读取存储器中某单元的数据,则CPU是主设备,而存储器是从设备。总线事务类型通常根据它的操作性质来定义,典型的总线事务类型有"存储器读""存储器写""I/O读""I/O写"和"中断响应"等,一次总线事务简单来说包括两个阶段:地址阶段和数据阶段。

突发传送事务由一个地址阶段和多个数据阶段构成,用于传送多个连续单元的数据,地址阶段送出的是连续区域的首地址。因此,一次突发传送事务可以传送多个数据。

3. 总线使用权

总线是由多个部件和设备所共享的,为了正确地实现它们之间的通信,必须有一个总线控制机构,对总线的使用进行合理的分配和管理。

主设备发出总线请求并获得总线使用权后,就立即开始向从设备进行一次信息传送。这种以主设备为参考点,向从设备发送信息或接收从设备送来的信息的工作关系,称为主从关系。主设备负责控制和支配总线,向从设备发出命令来指定数据传送方式与数据传送地址信息。各设备之间的主从关系不是固定不变的,只有获得总线使用权的设备才是主设备,如CPU等。但主存总是从设备,因为它不会主动提出要与谁交换信息的要求。

注意: *在定义总线数据传送操作是"输入"或"输出"时,必须以主设备为参考点,即从设备将数据送往主设备称为"输入",反之称为"输出"。这和前面提到的以主机为参考点的输入、输出的含义是不完全相同的。*

通常,将完成一次总线操作的时间称为总线周期。总线使用权的转让发生在总线进行一次数据传送的结束时刻。在一个总线周期开始时,对CPU或I/O设备的请求进行取样,并在这个总线周期进行数据传送的同时也进行判优,选择下一总线周期谁能获得总线使用权,然后在本周期结束时实现总线使用权的转移,开始新的总线周期。

7.1.2 总线的分类

1. 按功能层次分类

总线按功能层次分类如下。

(1) 片内总线

片内总线是芯片内部的总线,它是CPU芯片内部寄存器与寄存器之间、寄存器与ALU之间的公共连接线。片内总线在芯片内部,一般是看不见的。

(2) 系统总线

系统总线是计算机系统内各功能部件(CPU、主存、I/O 接口)之间相互连接的总线,系统总线也称为内总线,是构成计算机的主要组成部分。系统总线按传送的信息不同可以分为数据总线、地址总线和控制总线。

(3) 通信总线(外总线)

通信总线是用于计算机系统之间或计算机系统与其他系统(远程通信设备、测试设备)之间信息传送的总线。例如,一个计算机系统和另一个计算机系统,或者计算机系统与远程通信设备或测试设备之间的信息传送就通过外总线实现。

2. 按数据线的多少分类

按照总线中数据线的多少不同,总线又可分为以下几类。

(1) 并行总线

并行总线是含有多条双向数据线的总线,它可以实现一个数据的多位同时传输。并行总线具有数据传输率高的优点,但由于各条数据线的传输特性不可能完全一致,当数据线较长时,数据各位到达接收端时的延迟可能不一致,会造成传输错误。

(2) 串行总线

串行总线是只含有一条双向数据线或两条单向数据线的总线,串行总线可以实现一个数据的各位按照一定的速度和顺序依次传输。由于按位串行传输数据对数据线传输特性的要求不高,在长距离连线情况下仍可以有效地传送数据,所以串行总线的优势在于远距离通信。

7.1.3　总线的组成及性能指标

1. 总线结构

在单机系统中,从系统总线角度出发,总线的基本结构如下。

① 单总线结构:只有一条系统总线,所有部件通过系统总线接入。

② 双总线结构:在单总线的基础上增加一条专用于 CPU 和主存之间的数据传送通路。

③ 三总线结构:在双总线的基础上再增加一条 I/O 总线。

(1) 单总线结构

最简单的总线结构是单总线结构,如图 1-3 所示。各大部件都连接在单一的一组总线上,故将这个单总线称为系统总线。

(2) 双总线结构

单总线系统中各部件只能分时工作,这就使信息传送的吞吐量受到限制,为此设计了图 7-2 所示的双总线系统结构。这种结构保持了单总线系统简单、易于扩充的优点,但又在 CPU 和主存之间专门设置了一组高速的存储总线,使 CPU 可通过专用总线与存储器交换信息,减轻了系统总线的负担。

(3) 三总线结构

图 7-3 为三总线系统的结构图,它是在双总线系统的基础上增加 I/O 总线形成的。其

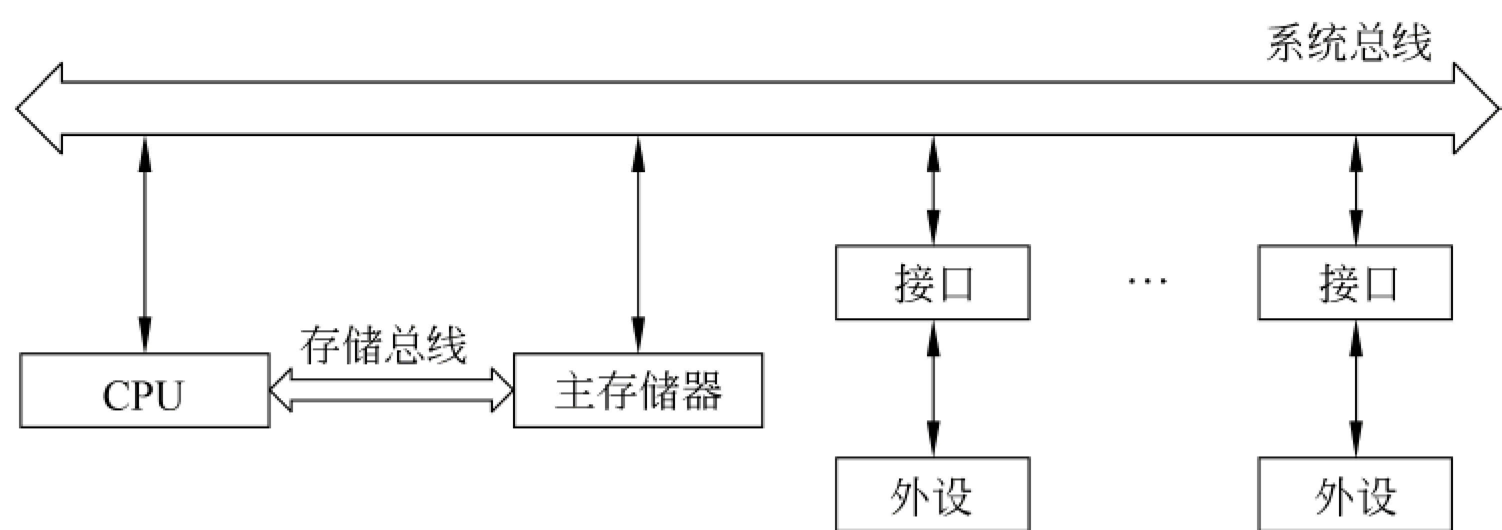


图 7-2　双总线结构

中，系统总线是 CPU、主存和通道(IOP)之间进行数据传送的公共通路，而 I/O 总线是多个外部设备与通道之间进行数据传送的公共通路。

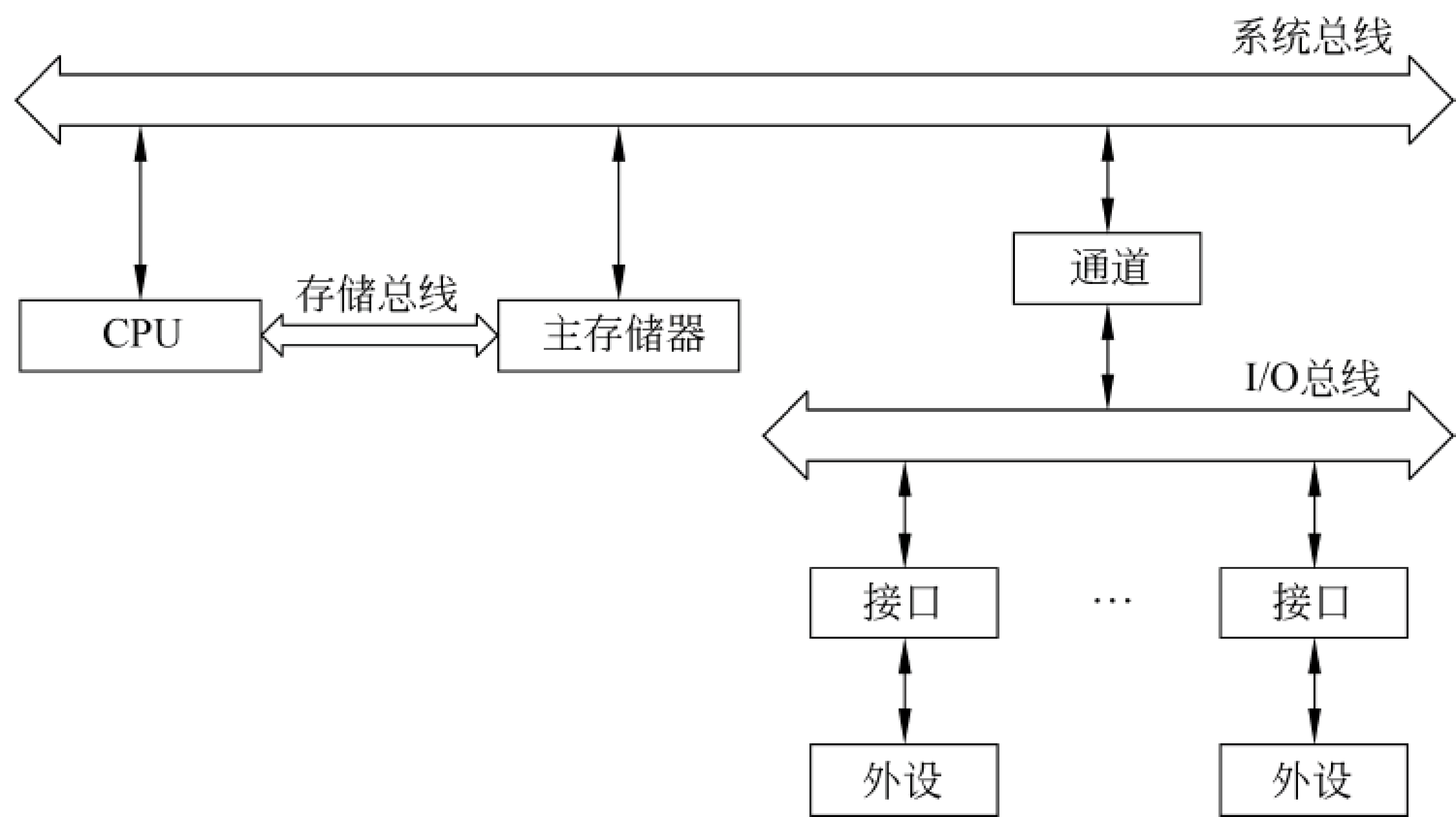


图 7-3　三总线结构

2. 总线特性

通常总线规范中会详细描述总线各方面的特性。

(1) 物理特性

物理特性又称机械特性，它规定了总线的线数、总线的插头、插座的形状、尺寸和信号线的排列方式等要素。

(2) 功能特性

功能特性描述总线中每一根线的功能。例如，CPU 发出的各种控制命令(如存储器读写、I/O 读写)、外设与主机的同步匹配信号、中断信号和 DMA 控制信号等。

(3) 电气特性

电气特性定义了每根线上信号的传递方向及有效电平范围。

(4)时间特性

时间特性规定了每根线在什么时间有效，以及不同信号之间相互配合的时间关系。只有规定了总线上各信号有效的时序关系，CPU 才能正确无误地使用。

3. 总线的性能指标

总线的主要性能指标如下。

(1) 总线宽度

总线宽度指的是总线的线数,它决定了总线所占的物理空间和成本。对总线宽度最直接的影响是地址线和数据线的数量,地址线的宽度指明了总线能直接访问存储器的地址空间范围,数据线的宽度指明了访问一次存储器或外设时能够交换的数据位数。

(2) 总线带宽

总线带宽定义为总线的最大数据传输率,即每秒传输的字节数。在同步通信中,总线的带宽与总线时钟密不可分,总线时钟频率的高低决定了总线带宽的大小。总线的带宽公式为:

$$B = W \times F / N$$

其中,B 为总线的带宽;W 为数据总线宽度,通常以字节为单位;F 为总线的时钟频率;N 为完成一次数据传送所用的时钟周期数。

(3) 总线负载

总线负载指连接在总线上的最大设备数量。大多数总线的负载能力是有限的。

(4) 总线复用

总线分时复用是指在不同时段利用总线上同一个信号线传送不同信号,例如地址总线和数据总线共用一组信号线。采用这种方式的目的是减少总线数量,提高总线的利用率。

(5) 总线猝发传输

猝发(突发)式数据传输是一种总线传输方式,即在一个总线周期中可以传输存储地址连续的多个数据。

除去以上提到的性能指标外,如总线是否具有即插即用功能,是否支持总线设备的热插拔,是否支持多主控设备,是否具有错误检测能力,是否依赖于特定 CPU 等也是评价总线性能的指标。

7.2 总线仲裁

总线是由多个部件和设备所共享的,连接到总线上的功能模块有主动和被动两种形态,CPU 可以作主方也可以作从方,而存取器模块只能用作从方。为了保证同一时刻只有一个申请者使用总线,总线控制机构中设置有总线判优和仲裁控制逻辑,即按照一定的优先次序来决定哪个部件首先使用总线,只有获得总线使用权的部件才能开始数据传送。总线判优按其仲裁控制机构的设置可分为集中式控制和分布式控制两种。

7.2.1 集中仲裁方式

总线控制逻辑集中在一处(如在 CPU 中)的称为集中式控制,就集中式控制而言,有下面 3 种常见的优先权仲裁方式。

1. 链式查询方式

链式查询方式如图 7-4 所示,总线控制器使用 3 根控制线与所有部件和设备相连,而 AB 和 DB 分别代表地址总线和数据总线。3 个控制信号如下。

总线请求(BR):该线有效,则表示至少有一个部件或设备要求使用总线。

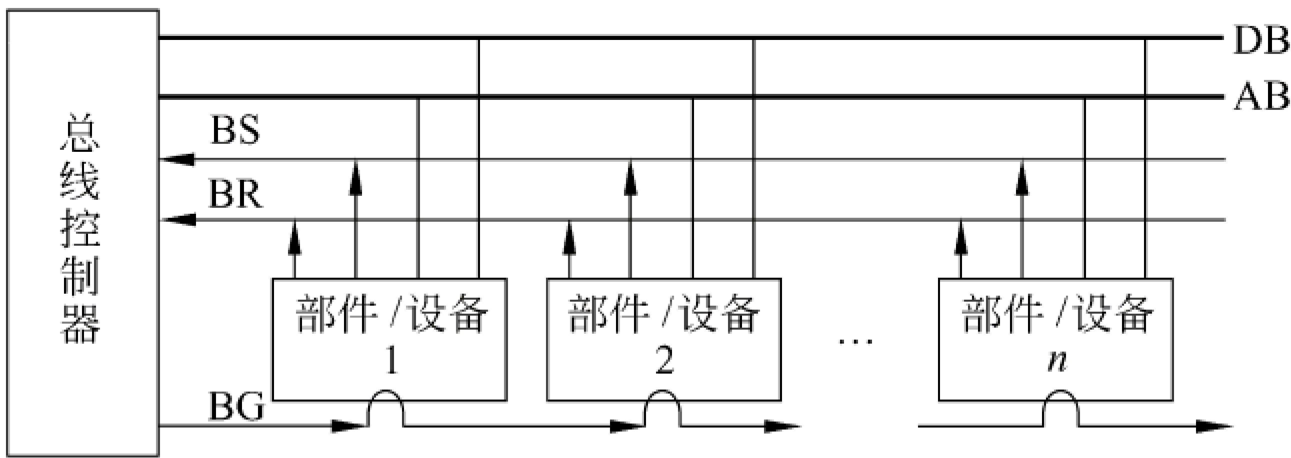


图 7-4　链式查询方式

总线忙(BS)：该线有效,则表示总线正在被某部件或设备使用。

总线批准(BG)：该线有效,则表示总线控制器响应总线请求。

与总线相连的所有部件经公共的 BR 线发出总线请求,只有在 BS 信号未建立前,BR 才能被总线控制器响应,并送出 BG 回答信号。BG 信号串行地通过每个部件,如果某个部件本身没有总线请求,则将该信号传给下一个部件;如果这个部件有总线请求,就停止传送 BG 信号,获得总线使用权。这时该部件将建立 BS 信号,表示它占用了总线,并撤销总线请求信号 BR,进行数据的传送。BS 信号在数据传送完后撤销,BG 信号也随之撤销。

显然,链式查询方式的优先次序是由 BG 线上串接部件的先后位置来确定的,在查询链中离总线控制器最近的设备具有最高优先权。

链式查询的优点是只用很少几根线就能按一定的优先次序来实现总线控制,并很容易实现扩充。缺点是对查询链的故障很敏感,如果第 i 个部件中的查询链电路有故障,那么第 i 个以后的部件都不能工作。另外,因为查询的优先级是固定的,所以若优先级较高的部件出现频繁的总线请求时,优先级较低的部件就可能会难以得到响应。

2. 计数器定时查询方式

计数器定时查询方式如图 7-5 所示,总线上的每个部件可以通过公共的 BR 线发出请求,总线控制器收到请求之后,在 BS 为 0 的情况下,让计数器开始计数,定时地查询各个部件以确定是谁发出的请求。当查询线上的计数值与发出请求的部件号一致时,该部件就使 BS 线置 1,获得了总线使用权,并中止计数查询,直至该部件完成数据传送之后,撤销 BS 信号。

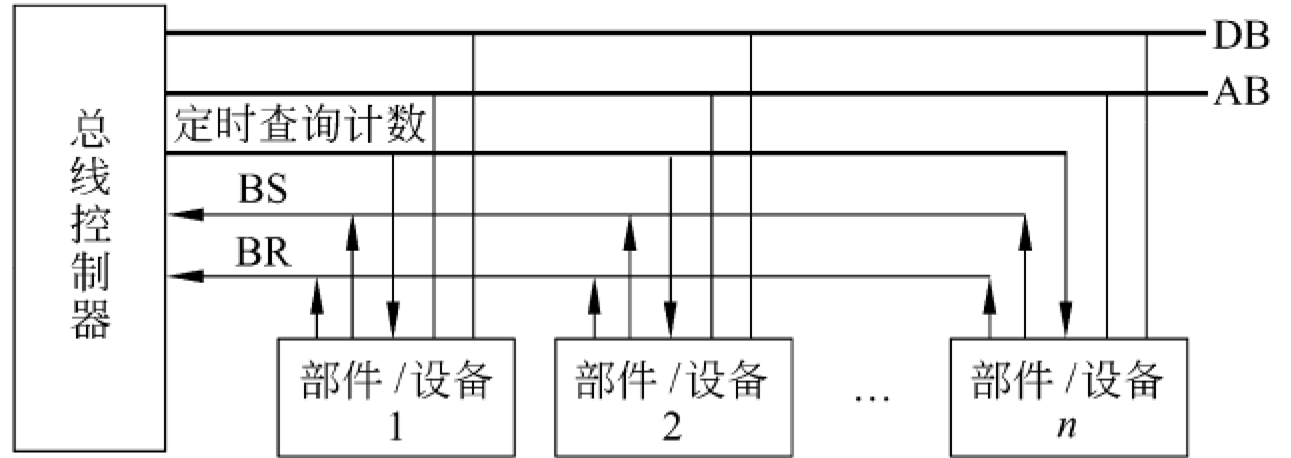


图 7-5　计数器定时查询方式

这种计数可以从 0 开始,也可以从中止点开始。如果从 0 开始,各部件的优先次序和链式查询方式相同,优先级的次序是固定的。如果从中止点开始,即为循环优先级,各个部件使用总线的机会将相等。计数器的初始值还可以由程序来设置,这就可以方便地改变优先

次序，增加系统的灵活性。定时查询方式的控制线数较多，对于 n 个部件，共需 $2+\lceil \log_2 n \rceil$根。

3. 独立请求方式

独立请求方式如图 7-6 所示，在这种方式中，每一个共享总线的部件均有一对控制线：总线请求 BR_i 和总线批准 BG_i。当某个部件请求使用总线时，便发出 BR_i，总线控制器中有一排队电路，根据一定的优先次序决定首先响应哪个部件的请求 BR_i，然后给该部件送回批准信号 BG_i。

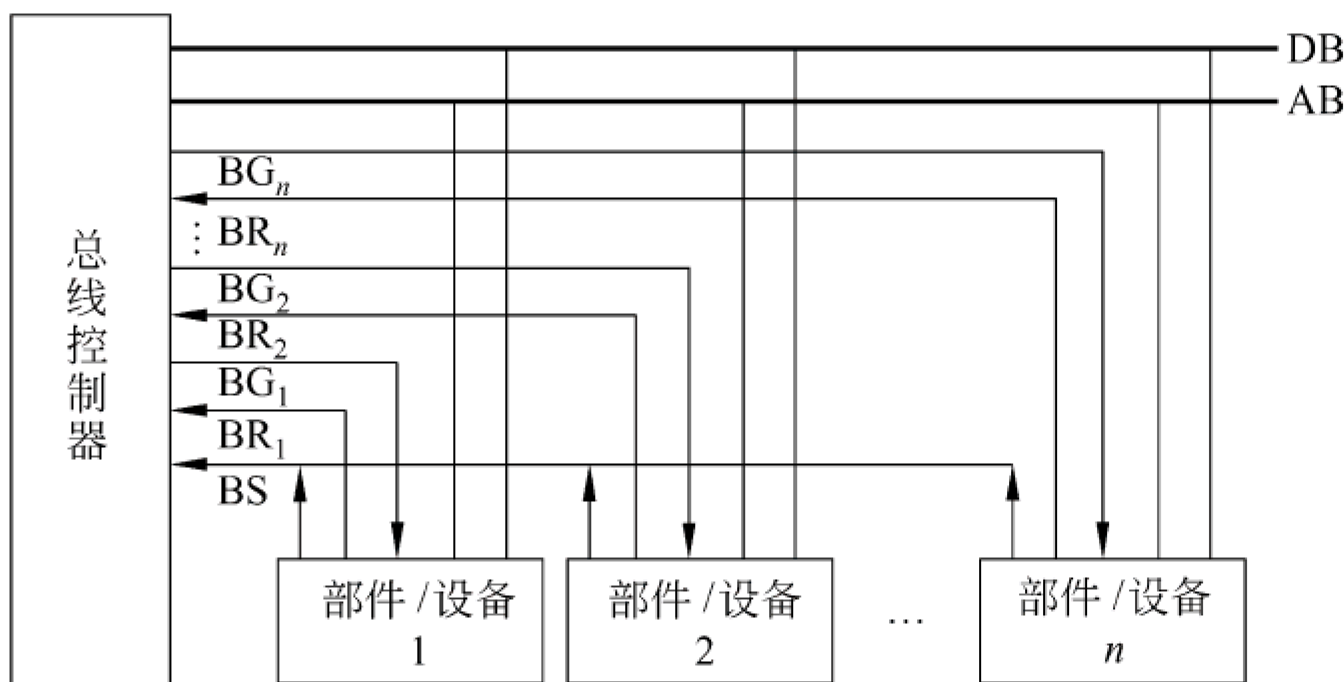


图 7-6　独立请求方式

独立请求方式的优点是响应时间快，然而这是以增加控制线数和硬件电路为代价的，对于 n 个部件，控制线的数目将达 $2n+1$ 根。此方式对优先次序的控制也是相当灵活的，它可以预先固定，也可以通过程序来改变优先次序。

7.2.2　分布仲裁方式

分布仲裁方式不需要中央仲裁器，即总线控制逻辑分散在连接于总线上的各个部件或设备中。连接到总线上的主方可以启动一个总线周期，而从方只能响应主方的请求。每次总线操作，只能有一个主方占用总线使用权，但同一时间里可以有一个或多个从方。对多个主设备提出的占用总线请求，一般采用优先级、冲突检测或公平策略等方法进行仲裁。其中，冲突检测仲裁方式一般用在网络通信总线上，具体来说就是"谈前先听，冲突重发"，主方在传输前，会侦听总线是否空闲，若空闲则立即使用总线；在传输过程中，还会侦听总线是否发生冲突，若发生冲突，则两个设备都会停止传输，延迟一个随机时间后再重新使用总线。

7.3　总线定时控制

主机与外设通过总线进行信息交换时，必然存在着时间上的配合和动作的协调问题，否则系统的工作将出现混乱。总线的定时控制方式一般分为同步方式和异步方式。

7.3.1　同步定时方式

所谓同步定时方式，是指系统采用一个统一的时钟信号来协调发送和接收双方的传送定时关系。时钟产生相等的时间间隔，每个间隔构成一个总线周期。在一个总线周期中，发

送和接收双方可以进行一次数据传送。由于是在规定的时间段内进行I/O操作,所以发送者不必等待接收者有什么响应,当这个时间段结束后就自动进行下一个操作。

同步方式中的时钟频率必须能适应在总线上最长的延迟和最慢的接口的需要。因此,同步方式的效率较低,时间利用也不够合理;同时,也没有办法知道被访问的外设是否已经真正地响应,故可靠性比较低。

7.3.2 异步定时方式

异步定时方式也称为应答方式。在这种方式下,没有公用的时钟,也没有固定的时间间隔,完全依靠传送双方相互制约的"握手"信号来实现定时控制。

通常,把交换信息的两个部件或设备分为主设备和从设备,主设备提出交换信息的"请求"信号,经接口传送到从设备;从设备接到主设备的申请后,通过接口向主设备发出"回答"信号,整个"握手"过程就是在一问一答中进行的。必须指出,从"请求"到"回答"的时间是由操作的实际时间决定的,而不是由CPU的节拍硬性规定的,所以具有很强的灵活性,而且对提高整个计算机系统的工作效率也是有好处的。

异步控制能保证两个工作速度相差很大的部件或设备之间可靠地进行信息交换,自动完成时间的配合。但是,异步控制较同步方式稍复杂一些,成本也会高一些。

异步方式根据"请求"和"回答"信号的撤销是否互锁,有以下3种情况。

(1) 不互锁

"请求"和"回答"信号都有一定的时间宽度,"请求"信号的结束和"回答"信号的结束不互锁,如图7-7(a)所示。

(2) 半互锁

"请求"信号的撤销取决于接收到"回答"信号,而"回答"的撤销由从设备自己决定,如图7-7(b)所示。

(3) 全互锁

"请求"信号的撤销取决于"回答"信号的来到,而"请求"信号的撤销又导致"回答"信号的撤销,如图7-7(c)所示。全互锁方式给出了最高的灵活性和可靠性,当然也付出了增加接口电路复杂性的代价。

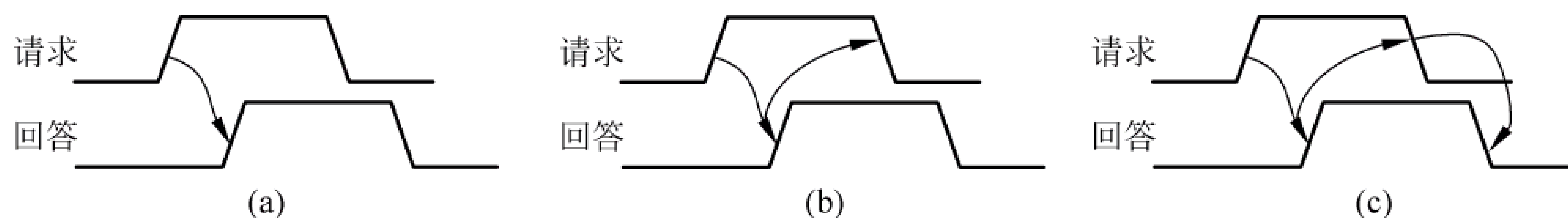


图7-7 请求与回答信号的互锁

7.4 总线标准

总线的标准制定通常有两种途径,一种是由具有权威性的国际标准化组织制定并推荐使用的,称为正式标准;另一种是由某个或某几个在业界具有影响力的设备制造商提出,而又被业内其他厂家认可并广泛使用的标准,即所谓的事实标准,这些标准可能需要经过一段

时间的使用，被厂商提供给有关组织讨论之后才能成为正式标准。

7.4.1　系统总线标准

通常，微机的系统总线都做成多个插槽的形式，各插槽引脚通过总线连在一起。总线接口引脚的定义、传输速率的设定、驱动能力的限制、信号电平的规定、时序的安排以及信息格式的约定等，都有统一的标准。下面对 PC 系列微机的总线标准进行简要介绍。

1. PC/XT 总线

PC/XT 总线是早期 PC/XT 微机所配备的系统总线，是 8 位总线标准。PC/XT 总线共有 62 个信号，是目前各类总线中最为精简的。

2. ISA 总线

ISA 总线最早用于 PC/AT 机，因而又称 AT 总线。它是在原 PC/XT 总线 62 线的基础上又增加一个 36 线的扩展槽，成为 16 位的总线标准，即工业标准体系结构（ISA），以支持 8/16 位的数据传送和 24 位寻址。

3. MCA 总线

IBM 公司于 1987 年推出了 32 位微通道结构（MCA）总线，并在 PS/2 机上使用。MCA 总线的数据线和地址线都扩展到 32 位，成为标准的 32 位扩展总线系统。但是，因为 MCA 总线与 ISA 总线不兼容，所以市场占有率不高。

4. EISA 总线

1988 年 9 月，由 Compaq 和 AST 等 9 家公司联合推出了一种既与 ISA 兼容，又在许多方面参考了 MCA 设计的总线标准，称为增强的工业标准体系结构（EISA），成为一种与 MCA 相抗衡的总线标准。

5. VL 总线（VESA 局部总线）

1992 年 5 月，视频电子标准委员会（VESA）制定了 VL 总线规范。VL 总线的数据宽度为 32 位，其主要优点是：协议简单、传输速率高、能够支持多种硬件的工作。但是，它的规范性、兼容性和扩展性较差。

6. PCI 局部总线

1991 年下半年，Intel 公司首先提出了 PCI 总线的概念，并与 IBM、Compaq、AST、HP 和 DEC 等公司联合，于 1993 年推出 PCI 总线。PCI 局部总线是一种高性能、32 位或 64 位地址数据线复用的总线，它的兼容性好，不受 CPU 品种的限制。

7. AGP

AGP（图形加速端口）是由 Intel 创建的新总线，专门用作高性能图形及视频支持。AGP 基于 PCI，且 AGP 插槽外形与 PCI 类似，但它有增加的信号，同时在系统中的定位不

同,是专门为系统中的视频卡设计的。

8. PCI-Express

PCI-Express(PCI-E)是最新的总线和接口标准,以取代几乎全部现有的内部总线(包括AGP和PCI),最终实现总线标准的统一。它的主要优势就是数据传输速率高,目前可高达10GB/s以上,而且还有相当大的发展潜力。PCI-E有1X到32X等多种规格,具有非常强的伸缩性,能满足现在和将来一定时间内出现的低速设备和高速设备的需求。PCI-E 16X用于取代目前的AGP接口,PCI-E 1X、2X用于取代目前的CPI接口,而针对服务器上的设备则有PCI-E 4X、8X、12X等。

PCI-E采用彻底的串行传输模式代替传统设备之间的并行传输模式,一个PCI-E串行连接称为lane,由两对单向传输的导线组成,一对负责发送,一对负责接收,每个周期虽然只传输1位信息,但它是以高达2.5Gb/s的速度传输。一个PCI-E可以由多个lane组成,标志为1X、2X等数字就是指有效的lane的总数。PCI-E 1X只需有4条线连接,16X则需要它的16倍即64条线连接,所以PCI-E 16X插槽要比PCI-E 1X插槽长得多。可以将PCI-E 1X的卡插在PCI-E 16X的插槽上(当然它依然只能工作在1X的速度下),但不能把PCI-E 16X的卡插在PCI 1X的插槽上。

PCI-E 1.0发表于2002年,传输速率为2.5GT/s;PCI-E 2.0则更进了一步,传输速率提升到了5GT/s;PCI-E 3.0将传输速率提升到8GT/s,并保持了对PCI-E 1.0/2.0的向下兼容,继续支持2.5GHz、5GHz信号机制。目前,PCI-E 4.0已经面市,传输速率达到16GT/s。PCI-E 5.0的规范也已出台。PCI-E规范的发展如表7-1所示。

表7-1 PCI-E规范的发展

版本号	传输速率	编码	带宽			
			×1	×4	×8	×16
PCI-E 1.0	2.5GT/s	8b/10b	250MB/s	1GB/s	2GB/s	4GB/s
PCI-E 2.0	5GT/s	8b/10b	500MB/s	2GB/s	4GB/s	8GB/s
PCI-E 3.0	8GT/s	128b/130b	1GB/s	4GB/s	8GB/s	16GB/s
PCI-E 4.0	16GT/s	128b/130b	2GB/s	8GB/s	16GB/s	32GB/s
PCI-E 5.0	32GT/s或25GT/s	128b/130b	4GB/s或3.1GB/s	16GB/s或12.5GB/s	32GB/s或25GB/s	64GB/s或50GB/s

PCI-E 1.0/2.0之所以数据传输率与原始传输率有差别,是因为它们采用的是8b/10b编解码机制;而PCI-E 3.0开始引入了128b/130b机制,可以确保几乎100%的传输效率,相比此前版本提升了25%,从而促成了传输带宽的翻番。

需要说明的是,采用8b/10b机制可使得发送的0、1数量保持基本一致,以保证直流平衡,即连续的1或0不超过5位,(每5个连续的1或0后必须插入一位0或1)。换句话说,8b/10b解码是将1组10位的输入数据经过变换得到8位数据位,即占用了20%的总带宽。通过8b/10b机制,可以保证传输的数据串在接收端能够被正确复原。8b/10b是目前许多高速串行总线采用的编码机制,例如USB 3.0、IEEE 1394b、SATA、PCI-E 1.0/2.0等。

7.4.2 外部总线标准

外总线是计算机系统之间互连的总线，通常使用标准的接口插头，其结构和通信规约也是标准的。

1. 串口和并口

串口又称通信口或 COM 端口，主要用于需要与系统进行双向通信的设备。这些设备包括早期的调制解调器、鼠标、扫描仪，以及所有向计算机发送信息和从计算机接收信息的其他设备。

传统的串口是异步传送，且面向字符的，它具有 20%的额外信息开销，这些信息是标识每个字符所需的。每个通过串行连接发送的字符都是由一个标准的起止信号来形成数据帧的，在每个字符前是一个独立的二进制"0"(称为起始位)，随后的 8 个二进制数字将组成数据的一个字节。在字符后跟有 1 个或 2 个二进制"1"(称为停止位)。在通信的接收端，对字符的识别是利用起止信号，而不是根据字符到达的时间来进行。图 7-8 表示字符数据的串行格式，7 位 ASCII 码连续传送，由最低有效数字位开始，而以奇偶校验位结束，起始位为低电平，停止位为高电平。

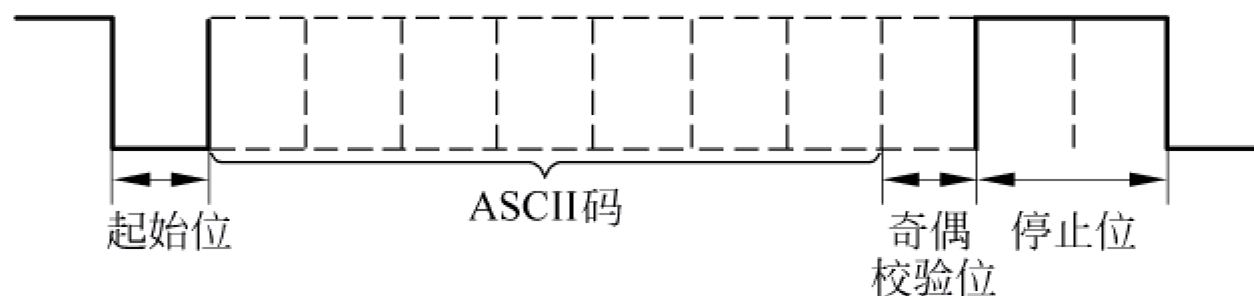


图 7-8　ASCII 码的串行格式

所有串口的核心是通用异步收发器(UART)芯片，该芯片可以实现将计算机输出的并行数据转换成串行格式，或者将串行数据转换成并行格式送回计算机的过程。当通信对象远离主机时，采用串行传送方式更经济、有效。RS-232 或 RS-422 等串行总线标准在计算机终端中广泛采用。

并口一般用于将打印机等设备连接到计算机上。并口之所以被称为并口，是因为它有 8 条数据线，可以通过这 8 条数据线同时发送包含数据的一个字节的所有数据位。

IEEE 1284 标准定义了并口的物理特性，标准中给出了 5 种不同的并口操作模式，如表 7-2 所示。

表 7-2　IEEE 1284 并口模式

并口模式	方向	传输率
半字节(4 位)	输入	50KB/s
字节(8 位)	输入	150KB/s
兼容	输出	150KB/s
EPP(增强型并口)	输入输出	500KB/s～2MB/s
ECP(扩充能力端口)	输入输出	500KB/s～2MB/s

早期计算机的并口仅能用来将信息从计算机发送到设备(输出)。后来出现的标准并口

能实现8位输出(称为兼容模式)和4位输入(称为半字节模式),标准并口的有效传输率是:输出150KB/s,输入50KB/s。目前的计算机系统,则往往带有功能更强大的并行接口,例如双向并口、EPP或ECP。双向并口使用标准的8位数据线可以实现8位输入和8位输出,且都能达到150KB/s的传输率。EPP或ECP被称为高速并口,它们可以获得2MB/s的传输率,可以用来支持诸如Zip驱动器、CD-ROM驱动器、扫描仪、磁带机,甚至硬盘等外设。

由传统的串口、并口完成的功能现在已经越来越多地由新的接口类型完成了。例如,高速串行总线USB和IEEE-1394都是远胜于标准串口和并口性能的高速通信接口,可以用来实现与高速外设的连接。

最新的高性能外部总线设计的趋势是使用串行结构,这样可以通过一根导线一次发送一位数据。而并行结构则需要同时使用8根、16根或更多的导线来发送数据位。在相同的时钟速度下,并行总线要快得多。但是,提高并行连接的时钟速度却要比提高串行连接的时钟速度难得多,这是因为虽然8位数据被发送器同时发送,但由于传输延迟的原因,当它们到达接收器时,可能会有一些数据位先于其他数据位到达。随着导线的延长,这种信号的不同步将会更严重,这使得数据不能以高的传输速率来进行传送。而使用串行总线由于一次仅发送一位数据,无须担心每一位数据的到达时间,因而可以极大地提高时钟频率。另外,在高时钟频率下,并行传输的数据线之间将相互干扰,而串行传输可以忽略这一干扰。

2. USB接口

USB(通用串行总线)是一种外设总线标准,它的设计为计算机的外设带来了即插即用功能。USB的出现不再需要专用的端口,也减少了I/O卡的使用(从而也减少了因添加新卡而重新配置系统的需要),大大节省了重要的系统资源。带有USB的计算机可以支持对外设的自动识别与设置,只需要将外设在物理上连接到计算机即可,而不需要重新启动或运行安装程序。USB在一台计算机上最多可以同时支持127台设备的运行。

USB给所有连接的设备提供电源,并支持热插拔,这意味不需要关掉计算机或重新启动系统就可以动态地插接设备。

过去主板中主要采用USB 1.1和USB 2.0,而目前的主板中主要采用USB 2.0和USB 3.0。各USB版本间能很好地兼容。USB 1.1版以12Mb/s(1.5MB/s)的速率运行,由于所有的设备只能共享1.5MB/s的带宽,这就意味着每增加一台活动的设备,总线就将慢一些。USB 2.0是USB 1.1的扩展,所有USB 1.1设备都可以在USB 2.0总线上工作。USB 3.1是最新的USB规范,它向下兼容USB 1.1、USB 2.0和USB 3.0标准。USB 3.0和USB 3.1都采用了9针脚设计,其中4个针脚和USB 2.0的形状、定义均完全相同。USB各版本的传输率如表7-3所示。

表7-3 USB的传输率

版本号	速率称号	最大传输速率	最大输出电流	编码
USB 1.0	低速	1.5Mb/s(192KB/s)	500mA	8b/10b
USB 1.1	全速	12Mb/s(1.5MB/s)	500mA	8b/10b

续表

版本号	速率称号	最大传输速率	最大输出电流	编码
USB 2.0	高速	480Mb/s(60MB/s)	500mA	8b/10b
USB 3.0	超速	5.0Gb/s(640MB/s)	900mA	8b/10b
USB 3.1	超速+	10Gb/s(1.25GB/s)	5A	128b/132b

USB 2.0 基于半双工二线制总线，只能提供单向数据流传输，而 USB 3.0 采用了四线制差分信号线，故而支持双向并发数据流传输。此外，USB 3.0 引入了新的电源管理机制，支持待机、休眠和暂停等状态。USB 3.1 还分出 Gen1 和 Gen2，所谓 USB 3.1 Gen 1 实际上就是原来的 USB 3.0，真正的新标准是 USB 3.1 Gen 2。

USB 是迄今为止最通用的外部接口，可以连接鼠标、键盘、打印机、扫描仪、摄像头、闪存盘、手机、数码相机、移动硬盘、外置光驱、USB 网卡等几乎所有的外部设备。

3. IEEE 1394 接口

IEEE 1394 也被称为 i.Link 或 Fire Wire，同 USB 一样，IEEE 1394 支持外设热插拔，并为外设提供电源。IEEE 1394 的最初版本被称为 1394a，传输速率为 100Mb/s、200Mb/s、400Mb/s，其升级版本 1394b 能提供 800 Mb/s 或更高的传输速度，甚至有望支持 1600Mb/s 和 3200Mb/s 的传输速率。IEEE 1394 构建在菊花链或树状的拓扑结构上的，它支持多达 63 台设备。另外，与 SCSI 一样，IEEE 1394 能够在同一条总线上支持不同速率的设备。

相比于 USB 接口，在 USB 1.1 时代，1394a 接口在速度上还是占据了很大优势的，但在 USB 2.0 推出后，1394a 接口在速度上的优势就不再明显了。同时，现在绝对多数主流的计算机并没有配置 1394 接口，要使用必须要购买相关的接口卡。但是，IEEE 1394 仍然有一定的市场，这是因为它的一个重要优点是不要求连接微机，可以用来直接将数字视频(DV)摄像机连接到 DV-VCR 进行磁带复制或编辑。

4. eSATA 接口

eSATA 的全称是 External Serial ATA (外部串行 ATA)，它是 SATA 接口(详见 8.3.4 节介绍)的外部扩展规范。换言之，eSATA 就是“外置”版的 SATA，它是用来连接外部而非内部 SATA 设备。例如，拥有 eSATA 接口，就可以轻松地将 SATA 硬盘与主板的 eSATA 接口连接，而不用打开机箱更换 SATA 硬盘。

eSATA 拥有极大的传输速度优势。例如，USB 2.0 的数据传输率可以达到 60MB/s，IEEE 1394 的数据传输率可以达到 50～100MB/s，eSATA 最高却可提供 384MB/s(3Gb/s)的数据传输速度，远远高于 USB 2.0 和 IEEE 1394。在实际应用上，受硬盘内部传输率及主板的制约，实际数据传输率可能介于 1.5Gb/s～3Gb/s 之间，但仍高于 USB 2.0 和 IEEE 1394 的传输速率，并且依然保持方便的热插拔功能，用户不需要关机便能随时接上或移除 SATA 装置，十分方便。

eSATA 是伴随 SATA 接口衍生出来的新一代外置设备接口，在 USB 3.0 没有出现之前，eSATA 接口的传输速度相比 USB 2.0 有着绝对的优势，而随着 USB 3.0 接口的普

及,eSATA接口遇到了挑战。目前来看,eSATA与USB 3.0接口在速度上不相上下,但eSATA只拥有传输数据的功能,并不能够给设备供电,而USB 3.0则完全不存在这个问题。当然USB 3.0也有它的劣势,即要将硬盘中的数据通过USB 3.0接口传输,则必须拥有桥接芯片,而这个芯片则可能在传输数据的过程中消耗传输速度。从各种因素以及今后的发展趋势来看,也许未来USB 3.0接口会越来越强势,而eSATA接口很有可能成为历史。

习　　题

7-1　假设地址线有20位,允许寻址的主存空间有多大?假设地址线有32位,允许寻址的主存空间又有多大?

7-2　假设总线的工作频率为22MHz,总线宽度为16位,问总线带宽是多少?

7-3　PCI总线的时钟频率为33MHz/66MHz,当该总线进行32/64位数据传送时,总线带宽各是多少?

7-4　假定某同步总线在一个时钟周期内传送一个4字节的数据,总线时钟频率为33MHz,求总线带宽是多少?如果数据总线宽度改为64位,一个时钟周期能传送2次数据,总线时钟频率为66MHz,则总线带宽为多少?提高了多少倍?

7-5　分析哪些因素影响带宽。

7-6　某总线时钟频率为66MHz,在一个64位总线中,总线数据传输的周期是7个时钟周期传输6个字的数据块。

(1) 问总线的数据传输率是多少?

(2) 如果不改变数据块的大小,而是将时钟频率减半,问这时总线的数据传输率是多少?

7-7　为什么要设立总线仲裁机构?集中式总线控制常用哪些方式?它们各有什么优缺点?

7-8　总线的同步通信和异步通信有何不同?试举例说明一次全互锁异步应答的通信情况。

第8章 外部设备

外部设备是计算机系统中不可缺少的重要组成部分，本章将介绍磁介质存储器的存储原理，常用磁介质存储设备和其他辅助存储设备以及常见的输入输出设备的工作原理。

8.1 外部设备概述

中央处理器（CPU）和主存储器（MM）构成计算机的主机。除主机以外，而又围绕着主机设置的各种硬件装置称为外部设备或外围设备。它们主要用来完成数据的输入、输出、成批存储以及对信息加工处理的任务。

8.1.1 外部设备的分类

外部设备的种类很多，从它们的功能及其在计算机系统中的作用来看，可以分为以下4类。

1. 输入输出设备

从计算机的角度出发，向计算机输入信息的外部设备称为输入设备；接收计算机输出信息的外部设备称为输出设备。

输入设备有键盘、鼠标、扫描器、数字化仪、磁卡输入设备、语音输入设备等。输出设备有显示设备、绘图机、打印输出设备等。

另外，还有一些兼有输入和输出功能的复合型输入输出设备。

2. 辅助存储器

辅助存储器是指主机以外的存储装置，又称后援存储器。辅助存储器的读写，就其本质来说也是输入或输出，所以可以认为辅助存储器也是一种复合型的输入输出设备。

目前，常见的辅助存储器有硬磁盘存储器、磁带存储器及光盘存储器等。

3. 终端设备

终端设备由输入设备、输出设备和终端控制器组成，通常通过通信线路与主机相连。终端设备具有向计算机输入信息和接收计算机输出信息的能力，具有与通信线路连接的通信控制能力，有些还具有一定的数据处理能力。

终端设备一般分为通用终端设备和专用终端设备两大类。专用终端设备是指专门用于某一领域的终端设备;通用终端设备则适用于各个领域,它又可分为会话型终端、远地批处理终端和智能终端。

4. 过程控制设备

当计算机进行实时控制时,需要从控制对象取得参数,而这些原始参数大多数是模拟量,需要先用模数转换器将模拟量转换为数字量,然后再输入计算机进行处理。而经计算机处理后的控制信息,需先经数模转换器把数字量转换成模拟量,再送到执行部件对控制对象进行自动调节。模数转换、数模转换设备均是过程控制设备,有关的检测设备也属于过程控制设备。

8.1.2 外部设备的地位和作用

外部设备是计算机和外界联系的纽带、接口和界面,如果没有外部设备,计算机将无法工作。随着超大规模集成电路技术的发展,主机的造价越来越低,而外部设备的价格在计算机系统中所占的比例越来越高。由此可见,外部设备在计算机系统中占据的地位变得越来越重要了。

外部设备在计算机系统中的作用可以分为4个方面。

1. 外部设备是人机对话的通道

无论是微型计算机系统,还是小、中、大型计算机系统,要把数据、程序送入计算机或要把计算机的计算结果及各种信息送出来,都要通过外部设备来实现。因此,外部设备成为人机对话的通道。

2. 外部设备是完成数据媒体变换的设备

人们习惯用字符、汉字、图形、图像等来表达信息的含义,而计算机内部却是以电信号表示的二进制代码。因此,在人机对话交换信息时,首先需要将各种信息变成计算机能识别的二进制代码形式,然后再输入计算机;同样,计算机处理的结果也必须变换成人们所熟悉的表示方式,这两种变换只能通过外部设备来实现。

3. 外部设备是计算机系统软件和信息的驻在地

随着计算机技术的发展,系统软件、数据库和待处理的信息量越来越大,不可能全部存放在主存中,因此,以磁盘存储器或光盘存储器为代表的辅助存储器已成为系统软件、数据库及各种信息的驻在地。

4. 外部设备是计算机在各领域应用的桥梁

随着计算机应用范围的扩大,已从早期的数值计算扩展到文字、表格、图形、图像和语音等非数值信息的处理。为了适应这些处理,各种新型的外部设备陆续被制造出来。无论哪个领域、哪个部门,只有配置了相应的外部设备,才能使计算机在这些方面获得广泛的应用。

8.2 磁介质存储器的性能和原理

磁介质存储器的存储过程是一种电磁转换的过程。常见的磁介质存储器(如磁盘、磁带等)就是利用磁记录原理制成的。

8.2.1 磁介质存储器的读写

1. 磁记录介质和磁头

(1) 磁记录介质

在磁介质存储器中,信息是记录在一薄层磁性材料上的,这个薄层称为磁层。磁层与所附着的载体称为记录介质或记录媒体。

载体是由非磁性材料制成的。根据载体的性质,又可分为软质载体和硬质载体。软质载体一般为聚酯薄膜材料,硬质载体一般为铝合金片。

(2) 磁头

磁头是磁记录设备的关键部件之一,是一种电磁转换元件,能把电脉冲表示的二进制代码转换成磁记录介质上的磁化状态,即电→磁转换;反过来,能把磁记录介质上的磁化状态转换成电脉冲,即磁→电转换。

读写时,按磁头与磁记录介质之间的接触与否,可分为接触式磁头与浮动式磁头两种。在磁带和软盘中,由于是软质载体,只能采用接触式磁头。接触式磁头的结构简单,但会因磨损而降低磁头与记录介质的使用寿命。在硬盘中,由于是硬质载体,必须尽量减少磨损(特别是记录区),故采用浮动式磁头。硬盘读写时,盘片高速地旋转,带动盘面表层气流形成气垫,使质量很轻的磁头浮起,与盘面之间保持一个极小的间隙,磁头不与盘面直接接触。

在读写过程中,磁记录介质与磁头之间相对运动,一般是记录介质运动而磁头不动。

2. 写入过程

磁头对记录介质的写入过程如图 8-1 所示。在写磁头线圈中通以一定方向的写电流,所产生的磁通将从磁头的头隙进入记录介质,然后流回磁头,形成一个回路,于是在磁头下方的一个局部区域被磁化,形成一个磁化单元(或称记录单元),磁通进入的一侧为 S 极,流出的一侧为 N 极。如果写电流足够大,可使磁化区的中心部分达到饱和磁化。当这部分介质移出磁头作用区后,仍将留下足够强的剩磁。在写磁头线圈中通以正、负两个不同方向的写电流,就会产生两种不同的剩磁状态,正好对应二进制信息的“1”和“0”。

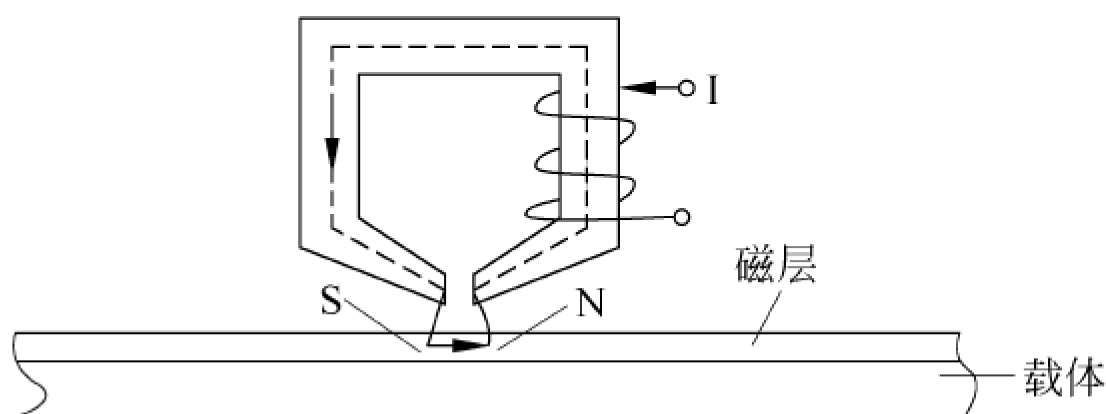


图 8-1 磁层的写入

3. 读出过程

读出时,读磁头线圈不外加电流。当某一磁化单元运动到读磁头下方时,使得磁头中流过的磁通有很大的变化,于是在读磁头线圈两端产生感应电动势 e。

$$e \propto -\frac{\mathrm{d}\phi}{\mathrm{d}t}$$

e 的极性与磁通变化的极性相反。当磁通 ϕ 由小到大变化时,在读磁头线圈中感应产生一个负脉冲;当磁通 ϕ 由大到小变化时,则感应产生一个正脉冲。上述脉冲信号经放大、检波、限幅、整形和选通后,获得符合要求的信号。

8.2.2 磁介质存储器的技术指标

衡量磁介质存储器的主要技术指标有下述几种。

1. 记录密度

记录密度又称存储密度,是指磁介质存储器上单位长度或单位面积所存储的二进制信息量。通常以道密度和位密度表示,也可用两者的乘积——面密度来表示。

(1) 道密度

道密度又称横向密度,是指垂直于磁道方向上单位长度中的磁道数目,道密度的单位是道/in(Tracks Per Inch,TPI)或道/mm(Tracks Per Millimeter,TPM)。

磁道指的是磁头写入磁场在记录介质上形成的磁化轨迹。为了避免干扰,磁道和磁道间需要保持一定的距离。相邻两条磁道中心线之间的距离称为道距。

(2) 位密度

位密度又称纵向密度,是指沿磁道方向上单位长度中所记录的二进制信息的位数,位密度的单位为位/in(bits per inch,bpi)或位/mm(bits per millimeter,bpm)。

2. 存储容量

存储容量是指整个磁介质存储器所能存储的二进制信息的总量,一般以字节为单位表示,它与存储介质的尺寸和记录密度直接相关。

磁介质存储器的存储容量有非格式化容量和格式化容量两种指标。非格式化容量是指磁记录介质上全部的磁化单元数;格式化容量是指用户实际可以使用的存储容量,也就是制造商给出的标称容量。格式化容量一般约为非格式化容量的 60%～70%。

3. 平均存取时间

在磁介质存储器中,当磁头接到读写命令后,从原来的位置移动到指定位置并完成读写操作的时间称为存取时间。对于采用顺序存取方式的多道并行读写的磁带存储器来说,没有寻找磁道的问题,故只需考虑磁头等待记录块的等待时间和信息的读写操作时间。对于采用直接存取方式的磁盘存储器来说,存取时间主要包括 4 个部分:第一部分是指磁头从原先位置移动到目的磁道所需要的时间,称为定位时间或寻道时间;第二部分是指在到达目的磁道以后,等待被访问的记录块旋转到磁头下方的等待时间,称为旋转时间或等待时间;第

三部分是信息的读写操作时间，也称为传输时间；最后是磁盘控制器的开销。由于寻找不同磁道和等待不同记录块所花的时间不同，所以通常取它们的平均值。传输时间和控制器的开销相对平均寻道时间 T_s 和平均等待时间 T_w 来说要小得多，所以磁盘的平均存取时间 T_a 约等于：

$$T_a \approx T_s + T_w = \frac{t_{smix} + t_{smax}}{2} + \frac{t_{wmix} + t_{wmax}}{2}$$

4. 数据传输率

磁介质存储器在单位时间内向主机传送数据的位数或字节数，称为数据传输率 D_r，单位为 b/s 或 B/s。数据传输率 D_r正比于记录密度 D 和磁记录介质通过磁头时的速度 v。

$$D_r = D \times v$$

式中：D——记录密度。对于单道存取的装置(如磁盘)为位密度；对于多道存储的装置(如磁带)则为位密度与磁道数之乘积。

v——速度。对磁带为走带速度；对磁盘为记录介质通过磁头时的线速度。

5. 误码率

误码率是衡量磁介质存储器出错概率的参数，它等于读出的出错信息位数和读出总的信息位数之比。

读出错误有硬错误和软错误之分。硬错误又称不可恢复的错误，它是由于记录介质上存在缺陷等原因引起的；软错误又称可恢复的错误，它是由偶尔落入记录介质和读写磁头之间的尘埃或电磁干扰引起的，可用重复的读操作来改正。

8.2.3 数字磁记录方式

为了提高磁介质存储器的性能，扩大存储容量，加快存取速度，除了要不断改善磁头和记录介质的电磁性能和机械性能之外，选用高性能的数字磁记录方式对提高记录密度和可靠性也是很重要的。

磁记录方式按照某种规律将一连串的二进制数字信息变换成记录介质上相应磁通翻转形式。磁记录方式可以分为直接记录方式和按位编码记录方式两大类，常见几种记录方式的写电流波形如图 8-2 所示，图中 T_0表示位周期。下面分别对它们进行讨论。

1. 直接记录方式

当记录密度较低时，可以不编码，直接按记录信息的“0”和“1”排序记录。这类记录方式有以下 3 种。

(1) 归零制(RZ)

记录“1”时，写磁头线圈中通以正向脉冲电流；记录“0”时，通以反向脉冲电流。由于脉冲电流均要回到零，故称为归零制。归零制的两个脉冲之间有一段间隔没有电流，相应的这段磁层未被磁化。

(2) 不归零制(NRZ)

记录“1”时，写磁头线圈中通以正向电流；记录“0”时，通以反向电流。由于磁头中电流

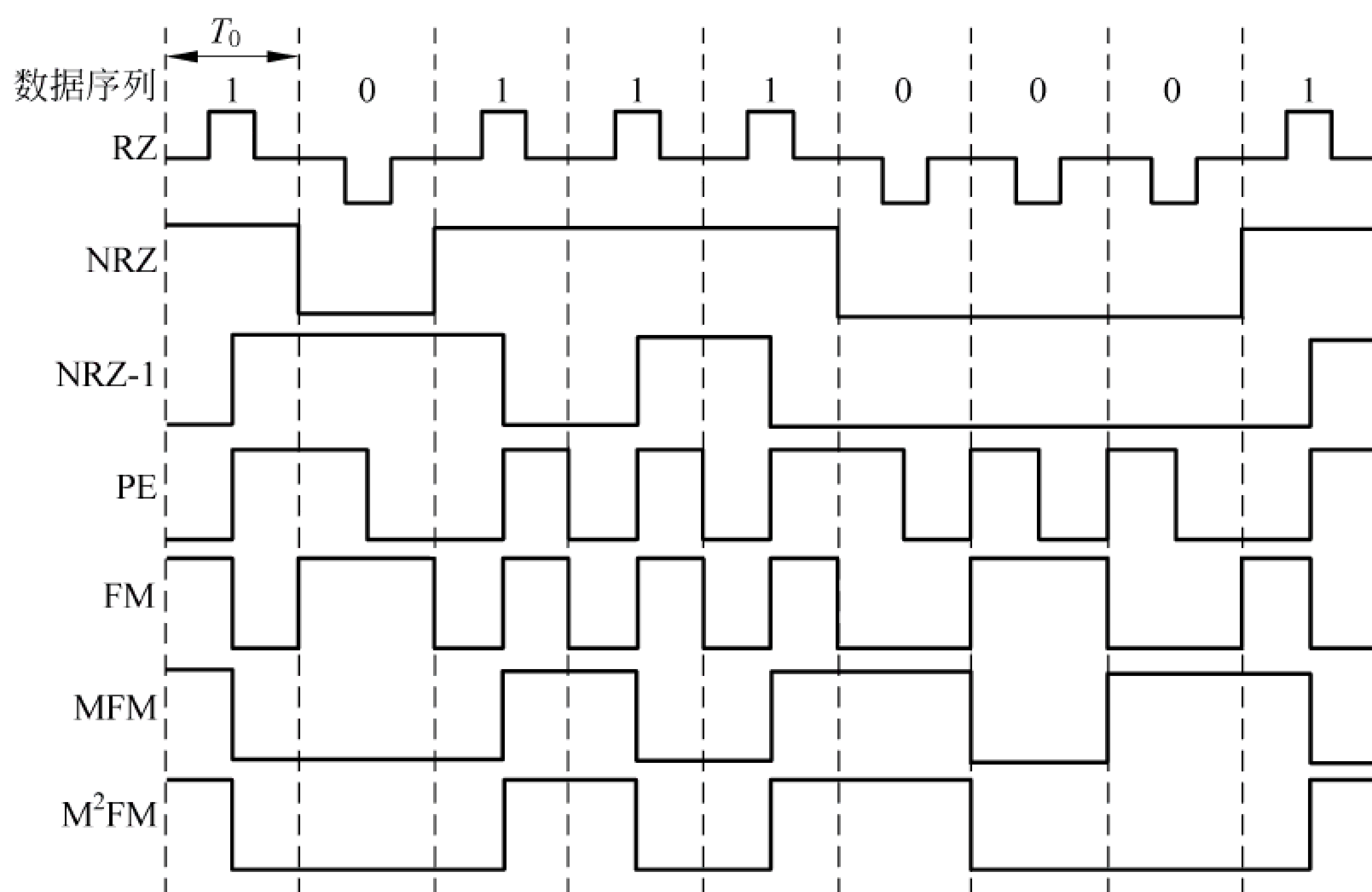


图 8-2　几种记录方式的写电流波形

不回到零,故称为不归零制。如果记录的相邻两位信息相同(即连续记录"1"或"0")时,写电流方向不变;只有当记录的相邻两位信息不相同(即"0"和"1"交替)时,写电流才改变方向,所以又称为异码变化或"见变就翻"的不归零制。

(3) 不归零-1 制(NRZ-1)

这是一种改进的不归零制,记录"1"时,在位周期中间写电流改变方向;而记录"0"时,写电流方向维持不变,所以称为见"1"就翻的不归零制。

2. 按位编码记录方式

下面几种记录方式都属于按位编码记录方式。

(1) 调相制(PE)

调相制又称曼彻斯特编码,它采用 0°和 180°相位的不同分别表示"1"或"0"。它的编码规则是:记录"1"时,写电流在位周期中间由负变正;记录"0"时,写电流在位周期中间由正变负。当连续出现两个或两个以上"1"或"0"时,为了维持上述原则,在位周期的边界上也要翻转一次。这种记录方式常用于磁带机中。

(2) 调频制(FM)

调频制是根据写电流的频率来区分记录"1"或"0"的。记录"1"时,写电流在位周期中间和边界各改变一次方向;记录"0"时,写电流仅在位周期边界改变一次方向。因此,记录"1"的磁通翻转频率为记录"0"时的两倍,故又称倍频制。若以 T_0 表示位周期,则调频制的磁通翻转间距为 $0.5T_0$ 和 T_0。这种记录方式主要应用于早期的硬磁盘机和单密度软磁盘机中。

(3) 改进的调频制(MFM)

MFM 制是在 FM 制基础上改进的一种记录方式,又称为延迟调制码或密勒码。其编码规则是:记录"1"时,写电流在位周期中间改变方向;记录单独的一个"0"时,写电流不改变方向;记录连续的两个"0"时,写电流在位周期边界改变方向。

改进的调频制的磁通翻转间距有 3 种：T_0、$1.5T_0$、$2T_0$，对应于 3 种不同的频率，所以又称为三频制。MFM 制的磁通翻转密度低于 FM 制，MFM 制的最小磁通翻转间距 $T_{min}=T_0$，而 FM 制的 $T_{min}=0.5T_0$，也就是 MFM 制可以减少 FM 制的磁通翻转数目，使之在相同数量的磁通翻转上存储两倍的数据。因此，采用 MFM 制的记录密度是 FM 制的两倍，这种记录方式主要应用于倍密度软磁盘机上。

(4) 改进的改进型调频制(M^2FM)

M^2FM 制是改进的 MFM 制方式。其编码规则是：记录“1”时，写电流在位周期中间改变方向；记录单独的一个“0”时，写电流不改变方向；记录连续的两个“0”时，写电流在第二个“0”起始的位周期边界处改变方向；记录连续多个“0”时，写电流在前两个“0”的位周期边界处改变方向；以后每隔两个“0”(假设其紧跟着的下一位还是“0”)，在位周期边界处写电流再改变一次方向。

改进的改进型调频制的磁通翻转间距有 4 种：T_0、$1.5T_0$、$2T_0$、$2.5T_0$，对应于 4 种不同的频率，所以又称为四频制。M^2FM 曾在软盘机和一些特殊用途的数字磁带机中使用。

3. 成组编码记录方式

除去上述 7 种记录方式外，还有一些成组编码方式，如群码制(GCR)、三位调制码(3PM)和游程长度受限码(RLL)等。它们是将数据序列中的数据位几位分成一组，然后按一定的变换规则变换成对应的记录码，再采用 NRZ-1 制写入记录介质，从而使记录密度得以提高。

现今硬盘中最流行的编码方式为游程长度受限(RLL)码，它的记录密度是调频制的 3 倍。游程长度受限码通常每次编码一组数据而不是单个数据。游程长度受限源于这些编码的两个主要特性，即两个实际的磁通转换之间允许的最小转换单元数目(游程长度)和最大的转换单元数目(游程受限)。该方式的不同变种使用不同的长度和受限参数，但只有两种真正得到普及：RLL2,7 和 RLL1,7。表 8-1 列出了 RLL2,7 的编码规则，它的编码严格按照 2→4、3→6、4→8 的规则进行变换。从表中可以看出两次实际的磁通转换 T 之间最小转换单元数目(N 的个数)是 2，最大转换单元数目是 7，所以称为 RLL2,7 码。

表 8-1　RLL2,7 数据到磁通转换的编码

数　据　位	磁 通 编 码	数　据　位	磁 通 编 码
10	NTNN	011	NNTNNN
11	TNNN	0010	NNTNNTNN
000	NNNTNN	0011	NNNNTNNN
010	TNNTNN		

注：T＝磁通翻转，N＝无磁通翻转。

如果最后一组的数据序列无法在表 8-1 中找到，可以通过添加多余的数据位以填补最后一组序列。

调频制与改进的调频制编码也可看作是游程长度受限编码的一种。调频制称为 RLL0,1，因为它用最少 0 个、最多 1 个转换单元区分两个磁通转换；改进的调频制称为 RLL1,3，因为它用最少 1 个、最多 3 个转换单元区分两个磁通转换。

8.2.4 编码方式的比较

尽管存在着很多种编码方式,但其中只有少部分仍然流行。近年最流行的3种基本编码方式是:

① 调频制(FM)。

② 改进的调频制(MFM)。

③ 游程长度受限(RLL)。

图8-3为用这3种不同的编码方式存储ASCII码字符“X”到硬盘上的写电流波形。

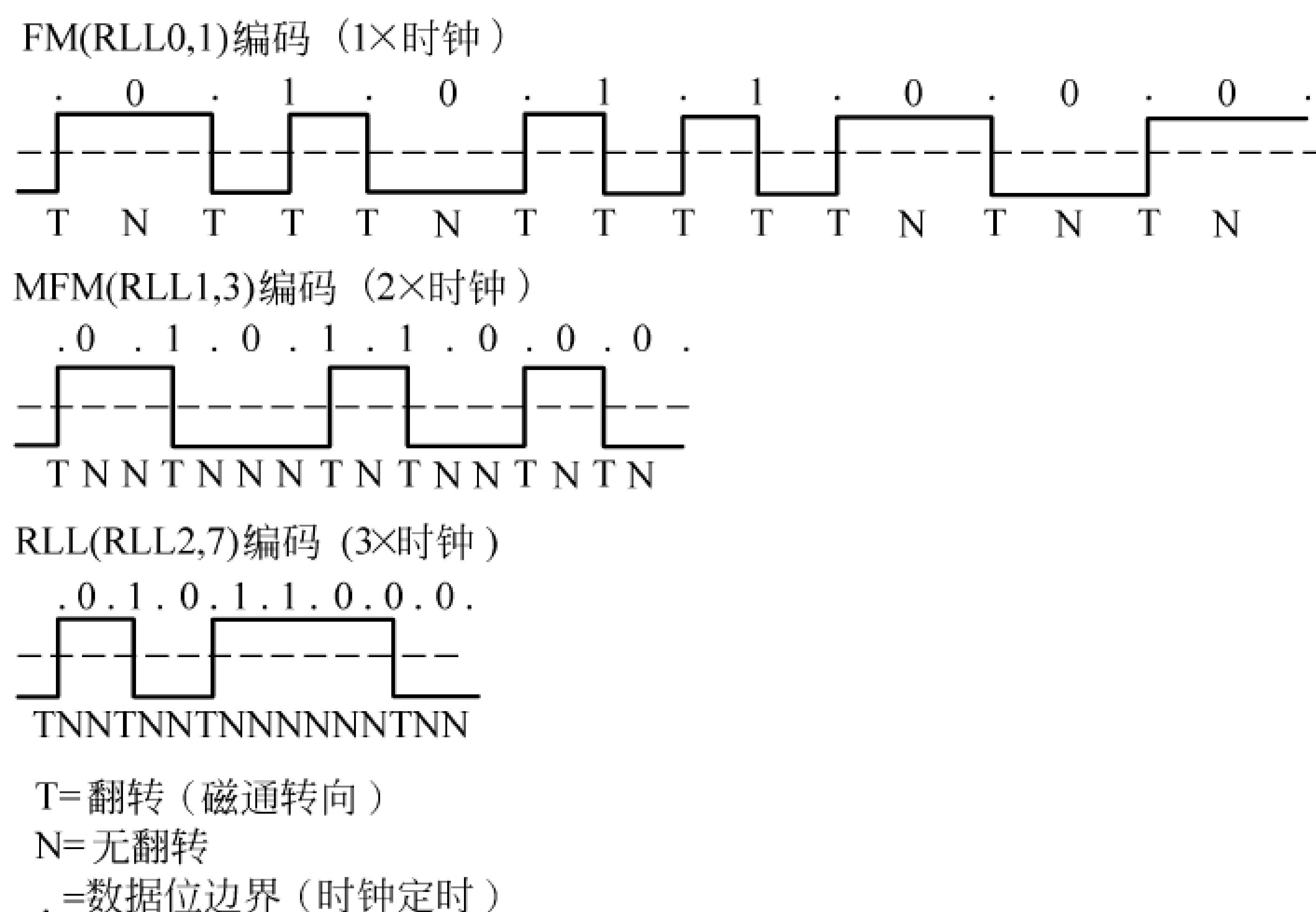


图8-3 用FM、MFM和RLL2,7编码ASCII字符“X”的写电流波形

在每种编码方式采样中,最上面表示各个位周期中所承纳的单个数据位01011000(字符“X”的ASCII码),位周期由时钟信号及时分开,如图表示为“.”;中间是实际的写电流波形;最下面表示磁通转换的情况,用T代表磁通翻转,用N代表无翻转。

FM制很容易解释。每个位周期有两个转换单元:一个代表时钟信息,另一个代表数据本身。所有的时钟转换单元都包含磁通翻转,而只有数据位为“1”时数据转换单元才包含磁通翻转,数据位为“0”时无磁通翻转。

MFM制的每个数据位也含有时钟和数据单元。但是,只有当一个“0”紧跟着另一个“0”存储时,时钟转换单元才含有一个磁通翻转。从左边开始,第一位为0,而前面的位未知(假设为0),因此磁通转换模式为TN;下一位为1,通常编码为转换模式NT;下一位为0,由于前面为1,因而存为NN。

RLL2,7由于是对成组的数据位编码而不是对单独的位编码,因而看起来比较复杂。从左边开始,第一组与表8-1中列出的组相匹配的是第一个3位数:010,这3位转换为磁通转换模式TNNTNN;下两位11可转换为TNNN,最后一组000可转换为NNNTNN结束。本例中无须添加多余的位填充最后一组。在本例中,任何两个磁通翻转T之间N的最小数目与最大数目为2和6,但在其他例子中最大转换单元数目可以达到7,这就是RLL2,7命名的由来。由于它记录的磁通翻转数目比MFM更少,因而时钟频率可增至FM的3倍,MFM的1.5倍,使得在相同的空间内存储更多的信息。

8.3　磁介质存储设备

磁介质存储器主要包括硬盘存储器、软盘存储器和磁带存储器。硬盘存储器的记录载体是硬质材料的，它的容量大、位价格低，是当今辅助存储器的主体。软盘存储器和磁带存储器的记录载体是软质材料的，软盘存储器在20世纪90年代还是微机辅助存储器的标准配置，但它的容量小、单位容量成本高、速度慢且可靠性差，随着计算机技术的发展，曾经应用最广泛的软盘现在已基本淡出人们的视线。

8.3.1　硬盘存储器的基本结构与分类

硬盘存储器具有存储容量大，使用寿命长，存取速度较快的特点。硬盘存储器的硬件包括硬盘控制器（适配器）、硬盘驱动器以及连接电缆。硬盘控制器（HDC）对硬盘进行管理，并在主机和硬盘之间传送数据；硬盘驱动器（HDD）中有盘片、磁头、主轴电机（盘片旋转驱动机构）、磁头定位机构、读写电路和控制逻辑等。新型的硬盘都已将控制器集成到驱动器单元中了。

为了提高单台驱动器的存储容量，在硬盘驱动器内使用了多个盘片，它们被叠装在主轴上，构成一个盘组；每个盘片的两面都可用作记录面，所以一个硬盘的存储容量又称为盘组容量。

根据头-盘是否是一个密封的整体，硬盘存储器可分为温彻斯特盘和非温彻斯特盘两类。温彻斯特盘的主要特点是磁头、盘片、磁头定位机构、主轴，甚至连读写驱动电路等都被密封在一个盘盒内，构成一个头-盘组合体。这个组合体不可随意拆卸，它的防尘性能好，可靠性高，对使用环境要求不高。而非温彻斯特式磁盘的磁头和盘片等不是密封的，因此要求有超净的使用环境。

根据磁头是否可移动，硬盘存储器可分为固定头硬盘和活动头硬盘两类。固定头硬盘机中，每个磁道对应一个磁头。工作时，磁头无径向移动，其特点是存取速度快，省去了磁头找磁道的时间，磁头处于加载工作状态即可开始读写。但是，由于磁头太多，使磁盘的道密度不可能很高，而整个磁盘机的造价却比较高。活动头硬盘机中，每个盘面上只有一个读写头，安装在读写臂上，当需要在不同磁道上读写时，要驱动读写臂沿盘面作径向移动。由于增加了寻道时间，所以其存取时间比固定头硬盘机要长。

8.3.2　硬盘驱动器

目前常用的硬盘驱动器都是活动头的温彻斯特盘，简称温盘。

1. 磁头

温盘的磁头采用接触启停式。所谓接触启停，是指在读写操作时磁头浮空，不与盘面记录区相接触，以免划伤记录区。但是，由于磁头的浮起要依靠盘片高速旋转时产生的气垫浮力，因此在启动前和停止后，磁头将仍与盘面接触。具体的做法是：在盘面记录区与轴心之间有一段空白区，被当作启停区或着陆区。未启动前及停止后，磁头停在启停区，与盘面接触。当盘片旋转并达到额定转速时，气垫浮力使磁头浮起并达到所需的浮动高度，然后将磁

头向外移至0号磁道,准备寻道。当读写工作完毕后,必须先将磁头移至启停区,盘片减速至静止,相应地磁头着陆,然后才能关机。

在读写时,磁头与盘面之间的间隙(又称飞高)极小,仅有0.2～0.5μm,甚至0.08μm。这样,磁头在通以写电流后,盘面的磁化单元很小,记录密度可以大大提高。

2. 磁头定位系统

磁头定位系统驱动磁头沿盘面径向移动寻道并精确定位。磁头定位系统应包括以下操作:

① 硬盘驱动器启动后,或是中途寻道出错后,要使磁头准确地回到0号磁道,以等待寻道命令。

② 要能快速、准确地将磁头移到指定磁道的中心位置。

③ 当硬盘驱动器发生故障或掉电后,要使磁头迅速退出盘面数据区,以保护盘面免受擦伤。

为了获得较高的道密度,定位系统必须非常精密;为了提高磁盘的寻道速度,定位系统的速度应尽量快。目前,在磁盘中采用的磁头定位系统有下述两种类型。

(1) 步进电机定位机构

在道密度不是很高的小容量磁盘中,一般采用步进电机驱动。整个定位机构是一个开环系统。根据现行磁道号与目的磁道号之差,求得步进脉冲数。每发一个步进脉冲,脉冲移动一个道距。步进电机定位机构的结构紧凑、控制简单,但定位精度比较低。

(2) 音圈电机定位机构

在道密度较高的磁盘中,多采用音圈电机驱动。音圈电机是线性电机,可以直接驱动磁头作直线运动,整个定位系统是一个带有速度和位置反馈的闭环调节自动控制系统,其特点是寻道速度快,定位精度高。

8.3.3 硬盘的信息分布和磁盘地址

1. 硬盘的信息分布

在硬盘中信息分布呈以下层次:记录面、圆柱面、磁道和扇区,如图8-4所示。

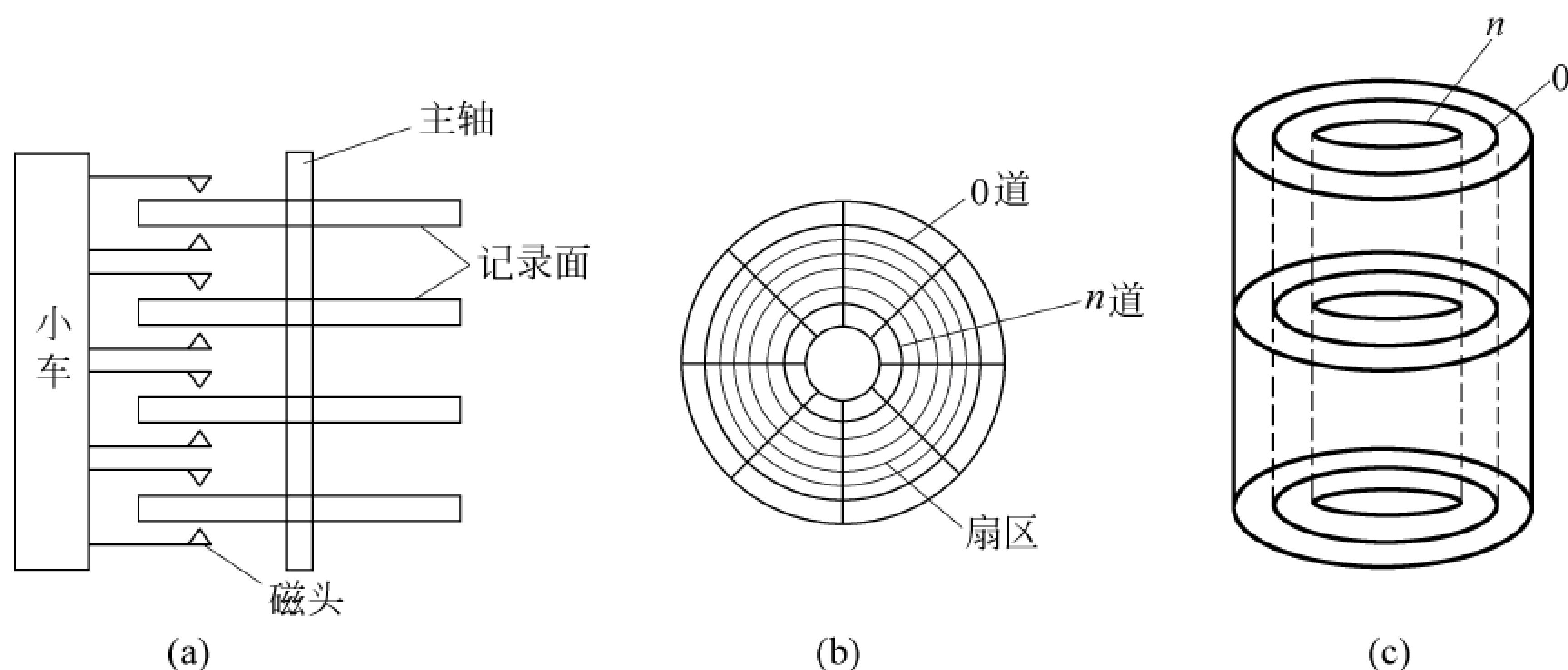


图8-4 磁盘信息分布示意图

(1) 记录面

一台硬盘驱动器中有多个盘片,每个盘片有两个记录面,每个记录面对应一个磁头,所以记录面号就是磁头号,如图 8-4(a)所示。所有的磁头安装在一个公用的传动设备或支架上,磁头一致地沿盘面径向移动,单个磁头不能单独地移动。

(2) 磁道

在记录面上,一条条磁道形成一组同心圆,最外圈的磁道为 0 号,往内则磁道号逐步增加,如图 8-4(b)所示。

(3) 圆柱面

在一个盘组中,各记录面上相同编号(位置)的诸磁道构成一个圆柱面,如图 8-4(c)所示。例如,某驱动器有 4 片 8 面,则 8 个 0 号磁道构成 0 号圆柱面,8 个 1 号磁道构成 1 号圆柱面……硬盘的圆柱面数就等于一个记录面上的磁道数,圆柱面号即对应的磁道号。

引入圆柱面的概念是为了提高硬盘的存储速度。当主机要存入一个较长的文件时,若一条磁道存不完,就需要存放在几条磁道上。这时应选择位于同一记录面上的几条磁道?还是选择同一圆柱面上的几条磁道呢?很明显,如果选择同一记录面上的不同磁道,则每次换道时都要进行磁头定位操作,速度较慢。如果选择同一圆柱面上的不同磁道,则由于各记录面的磁头已同时定位,换道的时间只是磁头选择电路的译码时间,相对于定位操作可以忽略不计,所以在存入文件时,应首先将一个文件尽可能地存放在同一圆柱面中。如果仍存放不完,再存入相邻的圆柱面内。

(4) 扇区

通常将一条磁道划分为若干个段,每个段称为一个扇区或扇段,每个扇区存放一个定长信息块(如 512 个字节),如图 8-4(b)所示。一条磁道划分多少扇区,每个扇区可存放多少个字节,一般由操作系统决定。磁道上的扇区编号从 1 开始,不像磁头或柱面编号从 0 开始。

2. 磁盘地址

主机向磁盘控制器送出有关寻址信息,磁盘地址一般表示为:驱动器号、圆柱面(磁道)号、记录面(磁头)号、扇区号。

通常,主机通过一个硬盘控制器可以连接几台硬盘驱动器,所以需送出驱动器号。调用磁盘常以文件为单位,故寻址信息一般应当给出文件起始位置所在的圆柱面号与记录面号(这就确定了具体磁道)、起始扇区号,并给出扇区数(交换量)。

8.3.4 硬盘存储器的技术参数

1. 硬盘的主要性能指标

(1) 硬盘容量

硬盘容量当然是越大越好。目前,微型计算机中的硬盘容量已经从数百吉字节发展到数千吉字节(即数个太字节),更大容量的硬盘还将不断推出。

硬盘的容量指标还包括硬盘的单碟容量。所谓单碟容量,是指硬盘单片盘片的容量。单碟容量越大,单位成本越低,平均访问时间也越短。目前,硬盘的单碟容量在 320GB 至

1TB 不等。

(2) 硬盘转速

硬盘主轴电机的旋转速度是决定硬盘内部传输率的关键因素之一,在很大程度上直接影响到硬盘的速度。硬盘转速以每分钟多少转(RPM)来表示,RPM 值越大,内部传输率就越快,访问时间就越短,硬盘的整体性能也就越好。最初,硬盘的主轴电机转速一般为3600RPM,现在硬盘的转速为5400RPM或7200RPM,高转速硬盘则可达到10 000RPM甚至15 000RPM。高转速可缩短硬盘的平均寻道时间和实际读写时间,但随着硬盘转速的不断提高也带来了温度升高、电机主轴磨损加大、工作噪音增大等负面影响。

(3) 道密度

因为盘片组是密封的、不可更换的,硬盘上的磁道密度可以非常高。今天的硬盘驱动器在介质上的道密度可达38 000TPI或更高。

(4) 平均存取时间

平均存取时间又称平均访问时间,如果忽略信息的传输时间和磁盘控制器的开销,则它是指磁头从起始位置到到达目标磁道位置,并且从目标磁道上找到要读写的数据扇区所需要的时间,包括了硬盘的寻道时间和等待时间,它体现了硬盘的读写速度。

硬盘的平均寻道时间是指硬盘的磁头移动到盘面指定磁道所需要的时间。这个时间当然越小越好,硬盘的平均寻道时间通常在8~12ms,而SCSI硬盘则应小于或等于8ms。

硬盘的等待时间又称潜伏期,是指磁头已处于要访问的磁道,等待所要访问的扇区旋转至磁头下方的时间。平均等待时间为盘片旋转一周所需要的时间的一半,一般在4ms以下。

(5) 缓存

缓存是硬盘控制器上的一块内存芯片,具有极快的存取速度,它是硬盘内部存储和外界接口之间的缓冲器。由于硬盘的内部数据传输率和外部数据传输率不同,缓存在其中起到一个缓冲的作用。缓存的大小与速度是直接关系到硬盘的传输速度的重要因素。当硬盘存取零碎数据时,需要不断地在硬盘与内存之间交换数据,如果缓存足够大,则可以将那些零碎数据暂存在缓存中,既减小了外系统的负荷,也提高了数据的传输速度。

早期的硬盘缓存基本都很小,只有几百千字节,而8~128MB已是现今主流硬盘的缓存设计,在服务器或特殊应用领域中还有缓存容量更大的产品甚至达到了256MB。

(6) 数据传输率

硬盘的数据传输率分为内部数据传输率和外部数据传输率。内部数据传输率也称为持续传输率,指的是磁头与硬盘缓存之间的数据传输率,它主要依赖于硬盘的旋转转速。外部数据传输率也称为突发数据传输率,指的是系统总线与硬盘缓存之间的数据传输率,外部数据传输率与硬盘接口类型和缓存大小有关。

2. 硬盘的接口标准

硬盘接口是硬盘与主机系统间的连接部件,不同的硬盘接口决定着硬盘与计算机之间的连接速度,在整个系统中,硬盘接口的优劣直接影响着程序运行速度的快慢和系统性能的好坏。从整体的角度上,硬盘接口分为IDE、SATA、SCSI和SAS等。

IDE接口也称ATA,或称并行ATA(PATA),采用16位数据并行传送方式。IDE接

口曾经具有相当辉煌的历史，从 ATA/33 发展到 ATA/100、ATA/133 等（ATA/133 代表外部数据传输率的理论最大值是 133MB/s）。然而，随着 SATA 接口的诞生，IDE 接口的硬盘已基本淡出了人们的视线。

SATA 是 Serial ATA 的缩写，即串行 ATA，这是一种完全不同于传统并行 ATA 的新型硬盘接口类型。与并行 ATA 相比，SATA 具有比较大的优势。首先，SATA 以连续串行的方式传送数据，可以使用较高的工作频率来提高数据传输的带宽。SATA 一次传送 1 位数据，能减少 SATA 接口的针脚数目。其次，SATA 的起点更高、发展潜力更大，SATA1.0 的数据传输率可达 150MB/s，这比最快的并行 ATA（即 ATA/133）所能达到的 133MB/s 最高数据传输率还高，SATA 2.0 的数据传输率已经高达 300MB/s，而 SATA 3.0 的数据传输率则更可高达 600MB/s。SATA 接口具备很强的纠错能力，还具有结构简单、支持热插拔的优点。几种磁盘接口的比较见表 8-2。

表 8-2　几种磁盘接口的比较

版　　本	接口速率/Gb/s	数据传输率/MB/s	数据线长度/m
SATA 3.0	6	600	2
SATA 2.0	3	300	1.5
SATA 1.0	1.5	150	1
PATA	1.33	133	0.5

在不同版本的 SATA 接口中，描述速率的单位有所不同，如 SATA 1.0 沿用了 PATA 的方法，单位为“MB/s”，SATA 2.0 之后单位为“Gb/s”，由于采用 8bit/10bit 编码，所以可以认为 300MB/s 就等于 3Gb/s。

SCSI 是小型计算机系统接口的缩写，它并不是专门为硬盘设计的接口，而是一种接入各种类型设备的通用快速接口。SCSI 接口具有应用范围广、多任务、带宽大以及热插拔等优点，但价格较高，主要应用于中、高端服务器和高档工作站中。Ultra160 SCSI 的数据传输率为 160MB/s，Ultra320 SCSI 的数据传输率为 320MB/s。

SAS（Serial Attached SCSI）即串行连接 SCSI，是新一代的 SCSI 技术，和现在流行的 Serial ATA（SATA）硬盘相同，都是采用串行技术以获得更高的传输速度，并通过缩短连线改善内部空间。SAS 是并行 SCSI 接口之后开发出的全新接口。此接口的设计是为了改善存储系统的效能、可用性和扩充性，并且提供与 SATA 硬盘的兼容性。和传统并行 SCSI 接口比较起来，SAS 不仅在接口速度上得到显著提升，传输速率高达 300MB/s，未来会达到 600MB/s 甚至更高，而且由于采用了串行线缆，可以实现更长的连接距离，还能够提高抗干扰能力。

3. 硬盘参数的计算

下面举例说明硬盘存储器参数的计算方法。设有一个盘面直径为 18 英寸（in）的磁盘组，有 20 个记录面，每面有 5 英寸的区域用于记录信息，记录密度为 100 道/英寸（TPI）和 1000b/英寸（bpi），转速为 2400r/min，道间移动时间为 0.2ms，试计算该盘组的容量、数据传输率和平均存取时间。

每一记录面的磁道数 N 为：

$$N=5\text{ 英寸/面}\times 100\text{ 道/英寸}=500\text{ 道/面}$$

最内圈磁道的周长为：

$$L=\pi\times(18-2\times 5)\text{英寸}=25.12\text{ 英寸}$$

以最内圈磁道的周长当作每条磁道的长度，故该盘组的存储容量(非格式化容量)为：

$$C=1000\text{b/英寸}\times 25.12\text{ 英寸/道}\times 500\text{ 道/面}\times 20\text{ 面}=251.2\times 10^6\text{b}=31.4\times 10^6\text{B}$$

磁盘旋转一圈的时间为：

$$t=\frac{1}{2400\text{r/min}}\times 60\text{s/min}=0.025\text{s}=25\text{ms}$$

数据传输率为：

$$D_r=\frac{\text{每一道的容量}}{\text{旋转一圈的时间}}=\frac{25120}{25}=1004.8\text{ b/ms}=1.0048\times 10^6\text{b/s}$$

$$=0.1256\times 10^6\text{ B/s}\approx 0.12\text{MB/s}$$

平均存取时间为：

$$T_a\approx\left[\frac{0+0.2\times 499}{2}+\frac{0+25}{2}\right]\text{ms}\approx 60\text{ms}$$

从上面的计算中可得到：

非格式化容量=最大位密度×最内圈磁道周长×总磁道数

新的磁盘在使用之前必须先进行格式化。格式化实际上就是在磁盘上划分记录区，写入各种标志信息和地址信息。这些信息占用了磁盘的存储空间，故格式化之后的有效存储容量要小于非格式化容量。它的计算公式为：

格式化容量=每道扇区数×扇区容量×总磁道数

8.3.5 硬盘的分区域记录

传统硬盘驱动器的每个磁道上记录的扇区数是相同的，因而存储的信息量也是相同的，这意味着在磁盘上位密度是变化的。因为内圈磁道的周长短，外圈磁道的周长长，所以内圈磁道的位密度高，外圈磁道的位密度低，最内圈磁道的位密度(最大位密度)决定了磁盘驱动器的容量。又因为每个磁道记录的信息量及转速是相同的，所以它们的数据传输率也是相同的。图8-5表示每个磁道有相同数量扇区的记录。

由于外圈磁道比内圈磁道要长一些，但存储的信息量却相同，所以外圈磁道上明显地存在着浪费。采用分区域记录(Zoned Recording)技术可以增加硬盘驱动器的容量。分区域记录就是把磁盘柱面分成一系列的组，又称区域(Zoned)。由于外圈磁道有更长的周长，所以外层磁道要比内层磁道包含更多的扇区，即外圈磁道上保存的信息比内圈磁道多。图8-6表示了采用分区域记录驱动器的磁盘扇区分布情况。

分区域记录的另一个影响是数据传输率随磁头所处的区域而变化。分区域驱动器还是以恒定速度旋转，可是由于外层区域每磁道有更多的扇区，所以数据传输速度要更快一些。这就是当今驱动器标注最小和最大连续传输速率的原因，因为传输速率取决于磁头读写的位置。

分区域记录技术的使用，大大地提高了硬盘利用率，与采用每磁道固定扇区的硬盘比较，使驱动器增加了20%～50%的硬盘容量。实际上，现在所有的硬盘都采用分区域记录。

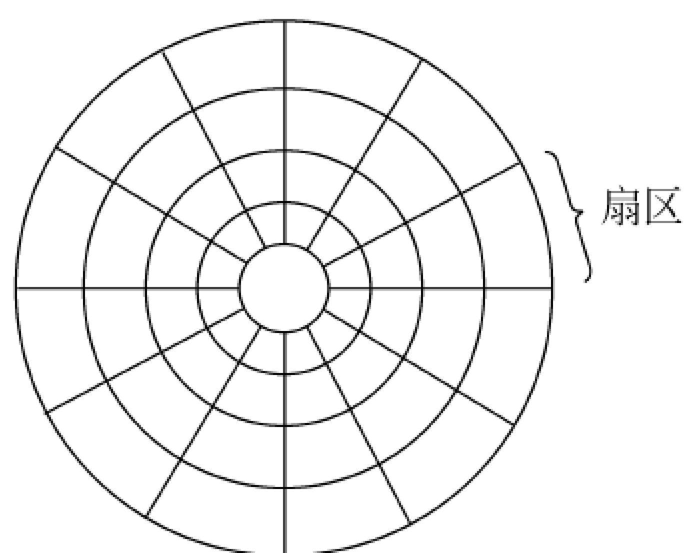


图 8-5 每个磁道有相同数量扇区

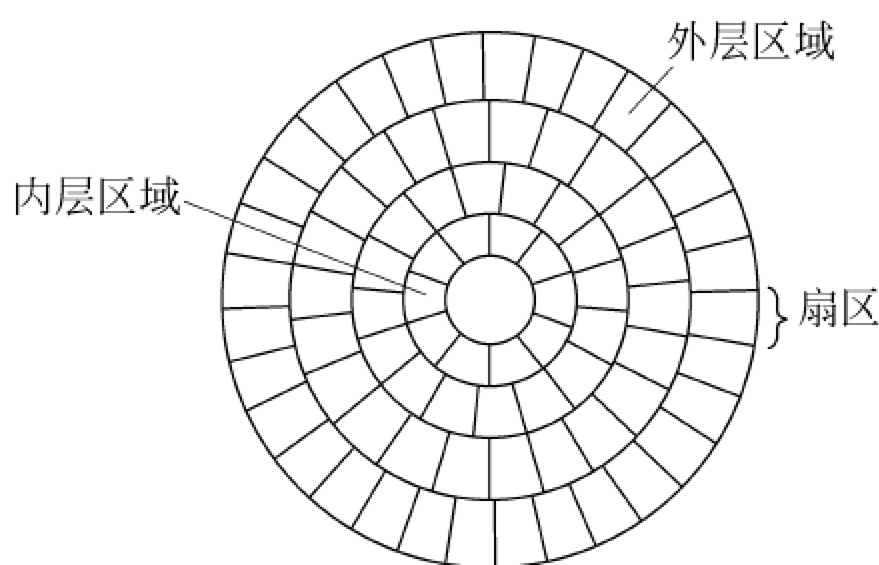


图 8-6 分区域记录磁盘扇区分布

8.3.6 硬盘的 NCQ 技术

NCQ(Native Command Queuing,全速命令排队)技术。它通过对内部队列中的命令进行重新排序实现智能数据管理,避免像传统硬盘那样机械地按照接收命令的先后顺序移动磁头读写硬盘的不同位置,从而减少了磁头反复移动带来的损耗,延长了硬盘的寿命。NCQ 技术是 SATA 2.0 规范中的重要组成部分。

根据磁盘地址,硬盘寻址的过程如下:

- 寻找目标圆柱面。
- 寻找目标盘面。
- 寻找目标扇区。

大多数情况下数据存入硬盘并非是顺序存入,而是随机存入,甚至有可能一个文件被分配在不同盘片上。对于不支持 NCQ 技术的硬盘来说,大量的数据读写需要反复重复上面的步骤,而对于不同位置的数据存取,磁头则需要更多的操作,这样降低了存取效率。而支持 NCQ 技术的硬盘对接收到的命令按照它们访问的地址的距离进行了重排列,对硬盘机械动作的执行过程实施智能化的内部管理,减少了磁头臂来回移动的时间。

例如,向硬盘先后下达了一组数据传送命令,按次序磁头可能会先读取 260 扇区,再读取 7660 扇区,然后又读取 261 扇区……如果对队列中的命令进行优化排列,可以先读 260 扇区,接着依次读 261 扇区,最后读取 7660 扇区……显然,命令重排列后减少了磁头臂来回移动的时间,使数据读取更有效。有效的排序算法除了考虑目标数据的线性位置,也会考虑其角度位置,并且还要对线性位置和角度位置进行优化,以使总线的服务时间最小,这个过程也称作“基于寻道和旋转优化的命令重新排序”。

8.4 磁 盘 阵 列

磁盘阵列(RAID)具有容量大、速度快、可靠性高、造价低廉的特点,它是目前解决计算机 I/O 瓶颈的有效方法之一,有着广阔的发展前景。

8.4.1 RAID 简介

RAID 是由美国加州大学伯克利分校的 D. A. Patterson 教授在 1988 年提出的。RAID 是 Redundant Array of Inexpensive Disks 的缩写,直译为“廉价冗余磁盘阵列”,简称为“磁

盘阵列”。后来RAID中的字母I被改作为Independent,RAID就成了“独立冗余磁盘阵列”,但这只是名称的变化,实质性的内容并没有改变。可以把RAID理解成一种使用磁盘驱动器的方法,它将一组磁盘驱动器用某种逻辑方式联系起来,作为逻辑上的一个磁盘驱动器来使用,图8-7为磁盘阵列外形。一般情况下,组成的逻辑磁盘驱动器的容量要小于各个磁盘驱动器容量的总和。RAID的具体实现可以靠硬件,也可以靠软件,Windows NT操作系统就提供软件RAID功能。



图8-7 磁盘阵列外形

RAID的优点如下:

① 成本低,功耗小,传输速率高。在RAID中,可以让很多磁盘驱动器同时传输数据,而这些磁盘驱动器在逻辑上又是一个磁盘驱动器,所以使用RAID可以达到单个磁盘驱动器几倍、几十倍甚至上百倍的速率。

② 提供容错功能。这是使用RAID的第二个原因,如果不考虑磁盘上的循环冗余校验(CRC)码,普通磁盘驱动器无法提供容错功能。RAID的容错是建立在每个磁盘驱动器的硬件容错功能之上的,所以它提供更高的安全性。

③ RAID比起传统的大直径磁盘驱动器来,在同样的容量下,价格要低许多。

8.4.2 RAID的分级

RAID可以分为7个级别,即RAID0～RAID6,如表8-3所示。在RAID1～RAID6的几种方案中,不论何时有磁盘损坏,都可以随时拔出损坏的磁盘后再插入好的磁盘(需要硬件上的热插拔支持),数据不会受损,失效盘的内容就可以很快地重建,重建的工作由RAID硬件或RAID软件来完成。但是,RAID0不提供错误校验功能,所以有人说它不能算作是RAID,其实这也是RAID0为什么被称为0级RAID的原因——0本身就代表“没有”。

表8-3 RAID的分级

RAID级别	名　称	数据磁盘数	可正常工作的最多失效磁盘数	检测磁盘数
RAID0	无冗余无校验的磁盘阵列	8	0	0
RAID1	镜像磁盘阵列	8	1	8
RAID2	纠错汉明码磁盘阵列	8	1	4
RAID3	位交叉奇偶校验的磁盘阵列	8	1	1
RAID4	块交叉奇偶校验的磁盘阵列	8	1	1
RAID5	无独立校验盘的奇偶校验磁盘阵列	8	1	1
RAID6	双维无独立校验盘的奇偶校验磁盘阵列	8	2	2

RAID级别的选择有3个主要因素:可用性(数据冗余)、性能和成本。如果不要求可用性,选择RAID0以获得最佳性能。如果可用性和性能是重要的而成本不是一个主要因素,则根据硬盘数量选择RAID1。如果可用性、成本和性能都同样重要,则根据一般的数据传输和硬盘的数量选择RAID3、RAID5。

RAID2～RAID6具有很强的功能,但毕竟太贵了,多用于高端服务器。后来又推出了

RAID 7,这是目前理论上性能最高的 RAID 模式,具有全新的 RAID 架构,由于其自身就带有实时操作系统和用于存储管理的软件工具,可完全独立于主机运行,且不占用主机 CPU 资源,所以 RAID 7 可以看作一种小型存储计算机,与其他 RAID 架构相比较更为先进,但价格非常昂贵。所以,下面只介绍 RAID0、RAID1,以及由它们派生出来的 RAID1+0 和 RAID0+1。

RAID0 是一个极端追求性能的方案,它至少基于两个硬盘,数据同时分布在各个硬盘上,没有容错能力,如图 8-8 所示。可以看出,由于一个传输过程由多个硬盘分担,也就相当于传输带宽增加,所以读写速度在 RAID 中最快。但是,因为任何一个硬盘损坏都会使整个 RAID 系统失效,所以安全系数反倒比单个的硬盘还要低。RAID0 是舍弃了 RAID 所强调的安全方面的要求而得到的极限性能,一般用在对数据安全要求不高但对速度要求很高的场合下。

RAID1 是为了达到 RAID 安全方面的极限而诞生的。每一个硬盘都有一个镜像的硬盘,镜像盘随时保持与原盘的内容一致,如图 8-9 所示。

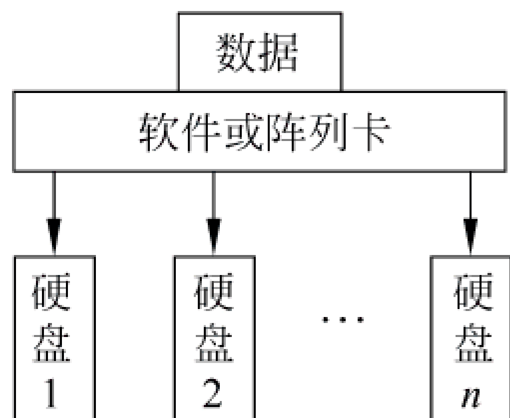


图 8-8 RAID0 数据存放

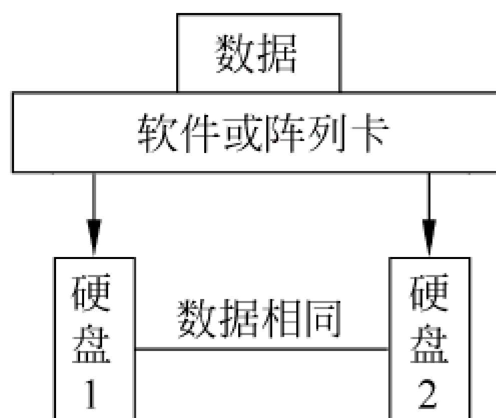


图 8-9 RAID1 数据存放

RAID1 具有最高的安全性,但只有一半的磁盘空间被用来存储数据。主要用在对数据安全性要求很高,而且要求能够快速恢复被损坏数据的场合。在这种方式下,不论原盘出现什么故障,都可以从镜像盘恢复回来,不过显然浪费了一半的磁盘空间,而且这种情况下唯一对提升性能有帮助的就是在读取时系统会同时从两个盘上搜索,把先读取到的数据传输回来。

虽然 RAID1 也可以获得少许的性能提升,但是相对 RAID0 来说恐怕是微不足道了。因此,在性能和安全兼顾的情况下,就出现了 RAID1+0 或 RAID0+1。RAID1+0 是先镜像再分组,而 RAID0+1 是先分组再镜像。RAID1+0 或 RAID0+1 都至少使用 4 个硬盘。RAID0+1 比 RAID1+0 有着更快的读写速度,但可靠性 RAID0+1 不及 RAID1+0,这是因为当 RAID1+0 有一个硬盘受损,其余 3 个硬盘会继续运作,而 RAID0+1 只要有一个硬盘受损,同组 RAID0 的另一个硬盘也会停止运作,只剩下两个硬盘运作。因此,RAID1+0 远比 RAID0+1 常用。

8.5 光盘存储器

相对于利用磁通变化和磁化电流进行读写的磁盘而言,用光学方式读写信息的圆盘称为光盘,以光盘为存储介质的存储器称为光盘存储器。

8.5.1 光盘存储器的类型

根据性能和用途的不同,光盘存储器可分为两类:一类是只读型光盘,如CD-ROM、DVD-ROM、BD-ROM等;另一类是可记录型光盘,如CD-R、CD-RW、DVD-RAM等。根据光盘结构的不同,光盘又可以分为CD、DVD、蓝光光盘几种类型。

1. CD光盘

CD光盘采用波长为780nm的红外激光读取和写入数据,其容量在700MB左右。只读的CD光盘和可记录的CD光盘在结构上并没有区别。

(1) CD-ROM

CD-ROM(Compact Disc Read Only Memory),即只读型光盘,又称固定型光盘。它由生产厂家预先写入数据和程序,使用时用户只能读出,不能修改或写入新内容。

(2) CD-R

CD-R光盘采用WORM(Write One Read Many)标准,光盘可由用户写入信息,写入后可以多次读出,但是只能写入一次,信息写入后将不能再修改,所以称为只写一次性光盘。

(3) CD-RW

这种光盘是可以写入、擦除、重写的可逆性记录系统。这种光盘类似于磁盘,可重复读写。

2. DVD光盘

DVD(Digital Versatile Disc)代表通用数字光盘,又称高容量CD。DVD光盘采用波长为650nm的红色激光读取和写入数据,其容量可以达到4.7GB。DVD光盘是在CD光盘之后的又一次重要的技术飞跃,它不仅在技术方面获得了很大的成功,并且迅速地推向市场,成为CD光盘的替代者。与CD光盘类似,DVD光盘也可以分为DVD-R、DVD-RW等。

3. 蓝光盘

蓝光盘(BD)利用波长较短(405nm)的蓝色激光读取和写入数据,并因此而得名。通常来说,波长越短的激光能够在单位面积上记录或读取更多的信息。因此,蓝光极大地提高了光盘的存储容量,容量可达25GB。蓝光盘是DVD光盘之后的下一代光盘格式之一,用以存储高品质的影音以及高容量的数据存储,在不久的将来高密度蓝光盘可能会取代DVD光盘。

8.5.2 光盘存储器的组成及工作原理

1. 光盘存储器的组成

光盘存储器由光盘控制器和光盘驱动器及接口组成。

光盘控制器主要包括数据输入缓冲器、记录格式器、编码器、读出格式器和数据输出缓冲器等部分。

光盘驱动器主要包括主轴电机驱动机构、定位机构、光头装置及电路等。其中,光头装

置部分最为复杂，是驱动器的关键部分。

光盘片是指整个盘片，盘片主要包括基板、记录层、反射层、保护层、印刷层5层。

基板一般采用聚碳酸酯晶片制成，是一种耐热的有机玻璃。无论是CD-ROM、DVD-ROM，还是CD-R、CD-RW光盘，表面上看都是一张120mm直径的盘片，中心有一个供固定用的15mm直径小圆孔，环孔中心半径13.5mm范围内和盘片外沿1mm内是空白区，真正存放数据的便是中间一段宽度为38mm的环形区域。

记录层又称染料层，以供激光记录信息。各种类型光盘的不同之处主要是记录层的化学成分存在差异。例如，CD-R光盘涂抹专用的有机染料，而CD-RW光盘则涂抹某种碳性物质。

反射层是反射激光光束的区域，借反射的激光光束读取光盘片中的信息。光线到达此层，就会反射回去。

保护层用来保护光盘中的反射层和记录层，防止信息被破坏。印刷层不仅可以标明光盘的内容，还可以起到一定的保护光盘的作用。

2. CD-ROM光盘的制作和读取

CD-ROM光盘是采用母盘灌制的方法大批量生产的。首先用事先编制好的程序控制激光刻片机，对一张玻璃基板进行蚀刻，将要存储的数据内容在玻璃基板上形成一个个数据凹痕，这个制作完成的玻璃基板就是大量压制CD-ROM光盘的模具。模具制造完成之后，用聚碳酸酯熔液倒入模具中，冷却后便变成具有同玻璃基板相应凹槽的基片，在其表面喷有一层厚度约为50nm的铝质反光涂料，其作用就是将读取数据的激光反射给接收装置；此外还必须覆盖一层起保护作用的透明基片，这样盘片的制作就完成了。

CD-ROM光盘上有一条从内向外的由凹痕和平坦表面相互交替而组成的连续的螺旋形路径，如图8-10所示。也就是说，数据和程序都是以刻痕的形式保存在盘片上的。当一束激光照射在盘面上，靠盘面上有无凹痕的不同反射率来读出程序和数据。因为程序和数据文件是按内螺旋线的规律顺序存放在盘上的，不能像磁盘驱动器那样读取文件的每个扇区，所以读出速度较慢。

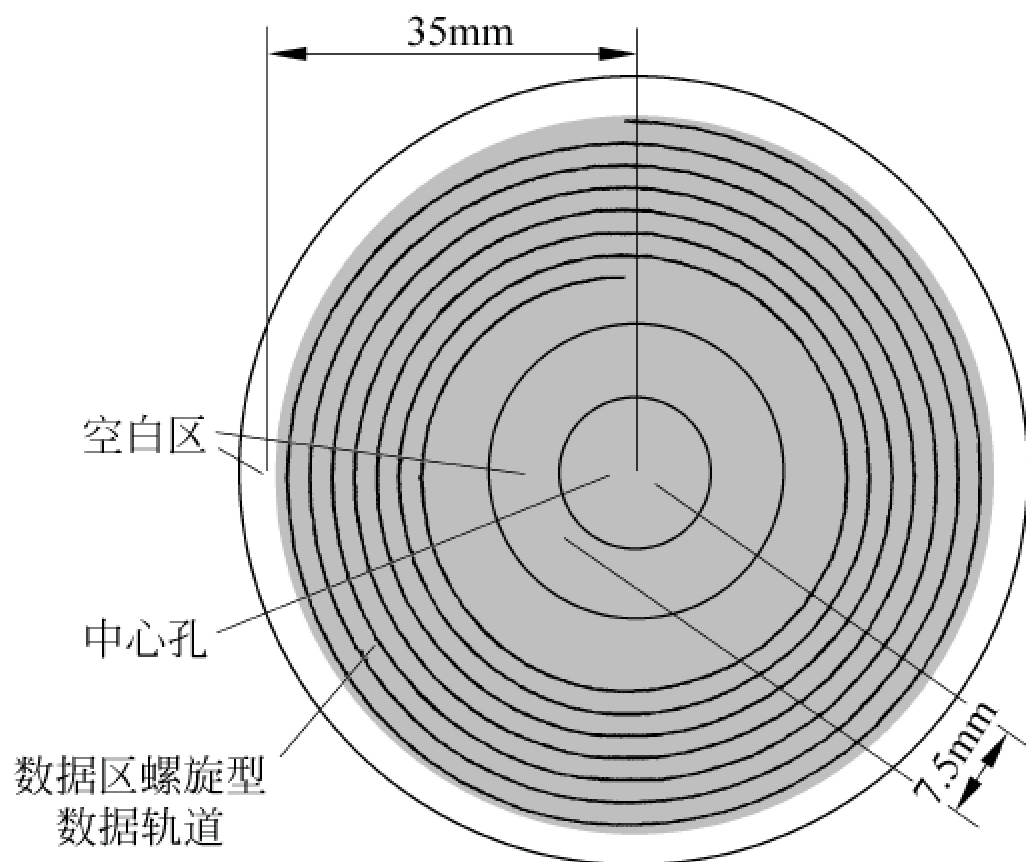


图8-10　CD-ROM光盘

当光驱读取这些盘片时,激光头射出的激光束在穿过表面的透明基片后,直接聚焦在盘片反射层上,被反射回来的激光会被光感应器检测到。每当激光通过凹痕时光强会发生变化,代表读取到数据“1”;而激光通过平坦表面时光强不发生变化,则代表读取到数据“0”。光驱的信号接收系统则负责把这种光强的变化转换成相应的电信号再传送到系统总线,从而实现数据的读取。

3. CD-R 光盘的读写原理

CD-R 光盘的写入是利用聚焦成 1μm 左右的激光束的热能使记录介质表面的形状发生永久性变化而完成的,所以只能写入一次,不能抹除和改写。

计算机送来的数据,先在光盘控制器内调制成记录序列,然后变成相应的记录脉冲信号。该脉冲信号在电流驱动电路内变为电流,送到激光器。激光器以 20mW 左右的功率发光,并聚焦成 1μm 左右的微小光点,落在记录介质表面上,CD-R 光盘上有一个有机染料刻录层,激光可以对该层的一个微小的区域加热,烧透染料层使其不透明,即打出一个微米级的凹坑。有凹坑代表写入“1”,无凹坑代表写入“0”。凹坑将永久性地保持现状意味着此种光盘只能一次写入。

读出时,用比写入功率低的激光束(约几毫瓦),连续照射在光盘上。由于有凹坑处的反射光弱,无凹坑处的反射光强,根据这一原理,当激光照射到光盘后,由光检测器将介质表面反射率的变化转变为电信号,经过数据检测、译码后送入计算机中,即可读出光盘上记录的信息。由于读出光束的功率仅是写入光束功率的 1/10,因此不会熔出新的凹坑。

CD-R 的盘片有金碟、绿碟、蓝碟 3 种,它们主要因记录层和反射层采用的材料不同而呈现出不同的颜色。

4. CD-RW 光盘的读写原理

CD-RW 光盘是利用激光照射引起记录介质的可逆性物理变化来进行读写的,光盘上有一个相位变化刻录层,所以 CD-RW 光盘又称为相变光盘。

相变光盘的读写原理是利用存储介质的晶态、非晶态可逆转换,引起对入射激光束不同强度的反射(或折射),形成信息一一对应的关系。

写入时,利用高功率的激光聚焦于记录介质表面的一个微小区域内,使晶态在吸热后至熔点,并在激光束离开瞬间骤冷转变为非晶态,信息即被写入。

读出时,由于晶态和非晶态对入射激光束存在不同的反射和折射率,利用已记录信息区域的反射与周围未发生晶态改变区域的反射之间存在着明显反差的效应,将所记录的信息读出。

擦除时,利用适当波长和功率的激光作用于记录信息点,使该点温度介于材料的熔点和非晶态转变温度之间,使之产生重结晶而恢复到晶态,完成擦除功能。

可写的 CD-R、CD-RW 的母盘制作过程大致是相同的,它们也都是采用激光刻片机蚀刻玻璃基板。不过因为没有存放数据,对玻璃基板不作凹槽的蚀刻,而只是利用程序的精密控制来刻出螺旋状轨迹。模具制造完成后再用聚碳酸酯生产塑胶基片,喷上铝或钛的反射涂层;为了实现数据写入,CD-R 和 CD-RW 盘片还必须再喷涂上一层对激光敏感的化学物质,当 CD-R 或 CD-RW 刻写数据时,高强度的激光会令这些物质发生物理变形或化学变性,产生许多存储数据的凹痕,以此实现数据的写入。

5. DVD光盘的工作原理

DVD采用与CD类似的技术,只是采用了波长更短的激光束(650nm)来读写数据。

DVD-ROM的读取过程与CD-ROM相似,从盘上读信息是将一个低能的激光束从光盘上各层的反射层反射回来的过程。激光从盘的下方发射一束激光,若该激光反射回来,光敏接收器就会感应到;如果激光遇到的是平地,它就会被反射回来;如果激光遇到的是凹陷不平的地方,就没有激光返回。

可写式DVD包括DVD-R、DVD-RAM、DVD-RW和DVD+RW,DVD-R是一种类似于CD-R的一次写介质,其他几种可写式使用了相位变化技术。

DVD每面可以有两层用来刻录数据,每一层单独压制,然后结合到一起最终形成1.2mm厚的光盘。与CD一样,DVD每一层都是以单一的螺旋形路径的形式印制,从光盘的最里端开始向外环绕。螺旋形路径上包含与CD中相同的凹痕和平地。每一层都覆盖一层反射激光的金属膜;外层的金属膜较薄,以便激光穿过它读取里层的数据。

6. BD光盘的工作原理

蓝色激光的波长仅为405nm,较小的光束聚焦更精确,最小凹坑长度只有0.15μm(DVD的最小凹坑长度为0.4μm),螺旋线的轨距只有0.32μm(DVD的轨距为0.74μm)。更小的光束、更小的凹坑以及更短的轨距结合起来,使得单层BD光盘的容量大约是DVD的5倍。

BD光盘的厚度与DVD大致相同,但两种光盘存储数据方式并不相同。在DVD中,数据存放在两个0.6mm基片上,可能会导致双折射问题,使光盘无法读取;而BD光盘将数据存放在1.1mm厚的基片上,可以防止双折射,克服了DVD的读取问题。

目前DVD的数据传输率为10Mb/s,BD光盘的数据传输率可达36Mb/s。

7. 光盘读取技术

CLV(Constant Linear Velocity)技术,即恒定线速度读取方式。这是在低于12倍速的光驱中使用的技术。它是为了保持数据传输率不变,而随时改变旋转光盘的速度。读取内道数据的旋转速度比外部要快许多。

CAV(Constant Angular Velocity)技术,即恒定角速度读取方式。它是用同样的速度来读取光盘上的数据。但光盘上的内道数据比外道数据传输速度要低,越往外越能体现光驱的速度。

CLV与CAV技术参数的比较如表8-4所示。而PCAV(Partial-CAV)技术是融合了CLV和CAV的一种技术,称为区域恒定角速度读取方式。它是在读取外道数据时采用CLV技术,在读取内道数据时采用CAV技术,从而提高了整体数据传输的速度。

表8-4 CLV与CAV技术参数的比较

技术参数	CLV(恒定线速度)	CAV(恒定角速度)
光驱转速	内道快于外道	恒定
数据传输率	恒定	外道高于内道
噪声水平	高	低

8.5.3 光盘驱动器

1. CD-ROM 驱动器

自1982年第一台CD-ROM驱动器问世以来,数据传输率已经成为其更新换代的标志。数据传输率是以150KB/s为基准成倍增加的。因此,习惯上把150KB/s传输率的光驱称为单倍速光驱,而把300KB/s传输率的光驱称为双倍速(2X)光驱,其后的四倍速(4X)光驱、八倍速(8X)、32倍速(32X)光驱等也由此而得名,如今市面上已有56倍速或更高的CD-ROM驱动器。

一个光驱仅仅是传输率高还不够,还应该配有足够的数据缓冲区,数据缓冲区大的光驱在读小型文件和随机文件时效果非常明显。

光驱的速度虽然在不断地"提升",但从某种角度而言,这只不过是光驱厂商玩弄的数字游戏而已。理论上讲32X光驱的速度应该是8X光驱的4倍,但实际速度的提升却并不明显,实际测试最多不超过2倍。这是什么原因呢?前面已经提到,12倍速以下的光驱一般采用CLV技术,在这种技术中,CD-ROM会根据现在正在读取的是光盘外道数据还是内道数据来控制电机以不同的角速度旋转光盘,读内道数据时光盘转速快,读外道数据时光盘转速慢,这样就能够保证在盘片的不同区域内保持恒定的数据传输率,并且对光盘的纠错性能也能有一定的提高。但是,在光驱速度进一步提高之后,如果仍采用CLA技术,电机将在高速的旋转下,不断地改变速度以保证在不同的内、外光道时线速度仍然相同,这将使电机的老化加剧。为了解决上述问题,对于12倍速以上的高速光驱采用了CAV技术。此时,光驱主轴的转速是恒定的,由于在光盘外道读取的数据要比内道多,所以外道的数据传输率高于内道的数据传输率。现在光驱厂商所标称的多少倍速,实际上是指在读取光盘外道时的最大速度,而从平均速度来看当然就没有那么高了。高倍速的光驱对只在内圈上有数据的光盘而言意义不大(很多光盘上的信息都并没有做满)。

2. CD-R 光驱

由于CD-R光驱可以对光盘写入,因此也称为光盘刻录机。光盘刻录机的速度有读取速度和写入速度,而后者才是刻录机的重要技术指标。在实际的读取和写入时,由于光盘的质量和烧录的稳定度下降,读取的速度会下降,烧录的速度也会下降。其次应该注意数据缓冲区的大小,缓冲区的大小是衡量刻录机的重要指标之一。因为在烧录时数据要先写入缓冲区再去烧录,如果缓冲区中的数据用完了,而后面的数据又没能及时补充上来,烧录当然就失败了。所以,缓冲区越大,烧录的成功率就越高,目前市场上光盘刻录机的缓冲区一般在512KB~4MB。

3. CD-RW 光驱

CD-RW光驱又称可擦写光盘刻录机,第一个CD-RW光驱标注为2/2/6,即其刻录速率为两倍速、重写速率为两倍速、读速率为6倍速,目前已经有20/10/40的版本出现。

CD-RW光驱可代替大部分的CD-R光驱,因为CD-RW光驱与CD-R完全兼容,并能以同样的能力读写通用的CD-R介质。CD-RW光盘的烧制或刻录的方式也与CD-R光盘

相同，主要区别在于它们可以擦除掉而多次重写，重写次数可以达1000多次。

4. DVD光驱

DVD光驱指读取或刻录DVD光盘的设备，事实上，任何DVD光驱都是CD光驱，即这类光驱既能读取CD光盘，也能读取DVD光盘。DVD除了密度较高以外，其他技术与CD-ROM完全相同。目前的DVD光驱多采用IDE或SATA接口，这意味着DVD光驱能像硬盘一样连接到主板的IDE或SATA接口上。

DVD光驱的实际旋转速度大约为同样倍速的CD光驱的3倍。许多DVD-ROM光驱列出了两个速度，一个是读取DVD盘的速度，另一个是读取CD盘的速度。例如，某DVD-ROM光驱的速度为16X/40X，这分别指的是读DVD和CD盘的性能。

5. BD光驱

BD光驱是能读取或刻录蓝光光盘的光驱，向下兼容DVD、CD格式。BD光驱有内置和外置之分，内置BD光驱多采用SATA接口，外置BD光驱则采用USB接口。

8.6 新型辅助存储器

随着操作系统和应用软件的逐渐庞大，需要更多的空间来存储它们及其创建的数据。除去前述的磁介质存储器和光存储器，近年来又出现了许多新型的辅助存储器，这些存储器的共同特点是容量大、可移动、使用方便。所谓可移动，是指存储器可随身携带，便于存储，可以在不同终端间移动，常见的可移动存储器使用磁介质、磁光介质或电子器件来存储信息。

可移动存储器可以存储若干个数据文件或不常使用的程序，也可以存储整个硬盘的内容。除了备份，它们还可以非常容易地将庞大的数据文件从一台计算机传递到另一台计算机中，或者用户可以将机密数据装入可移动存储器并将其带离办公室，以防泄露。

8.6.1 基于磁或磁光介质的可移动存储器

基于磁介质可移动存储器有两种基本类型：磁盘和磁带。磁盘介质的价格相对较贵，其容量一般来说也相对较小，在基于文件的系统中更容易使用，在复制少量文件时比较快，但在复制大量文件或者整个驱动器时则比较慢。磁带介质的价格总体来说比较便宜，其总容量也比较大，在图像或多文件系统中使用比较方便，用它来备份整个硬盘上的所有应用程序和数据非常合适，即适合于巨量备份，但复制单个文件时就显得比较费事了。

有两类常用的可移动磁盘驱动器，分别基于磁介质和磁光介质。磁介质驱动器采用与软盘或硬盘驱动器非常相似的技术，对数据进行编码和存储。磁光介质驱动器在盘上对信息进行编码时，使用了磁和激光相结合的新技术。

1. 移动硬盘

绝大多数移动硬盘尺寸为2.5in或3.5in，只有很少部分是微型硬盘(1.8in)。目前，移动硬盘的容量已经很大(可以从几百吉字节到几个太字节)，随着技术的发展，移动硬盘容量

越来越大,体积越来越小。

移动硬盘大多采用 USB、IEEE 1394 或 eSATA 接口,能提供较高的数据传输速度。一般移动硬盘由一个 USB 接口供电就可以了,但也有些移动硬盘使用一根 3 个分支的数据线,其中一个分支用于供电不足时补充供电。通常,移动硬盘应具有防震功能,在剧烈震动时盘片自动停转并将磁头复位到安全区,防止盘片损坏。

2. 大容量软盘

传统的软盘由于容量小、速度慢、不稳定,已逐步退出市场,在此之后大容量软盘(主要有 Zip 和 LS-120)开始争夺市场的主导地位。但是,随着传统 3.5 in 的软盘的完全消失,这两种大容量软盘存在的必要性也基本消失了。不过它们毕竟在辅助存储器的发展史上有过痕迹,所以在此还是为它们留下一笔记忆。

第一代 Zip 磁盘的容量约 100MB,是传统软盘容量(1.44MB)的 70 倍;使用并行接口;Zip 驱动器采用硬盘磁头技术(磁头不接触式读写),读写速度快,是传统软盘驱动器的 20 倍;最特别的是,每张 Zip 盘片都有密码设置功能。随着技术的发展,到 Zip 750 已达到了 750MB 的容量,使用 USB 2.0 接口,无论是容量还是速度都得到了巨大的改善。但遗憾的是,它不能兼容传统的软盘,导致 Zip 磁盘应用受到局限。

LS-120 磁盘存储容量为 120MB,而且 LS-120 驱动器可以向下兼容传统的软盘驱动器。在同样的盘片面积上能达到如此高的容量,这是因为采用了一些新的技术,例如:

① 光学定位技术。传统的软驱使用磁性定位,准确度不高,1.44MB 软盘只有 80 条磁道,而采用激光光学定位技术之后,在相同面积内划分出更多的磁道,可以达到 1736 条磁道。

② ZBR(圆周位记录)技术,也就是前述的硬盘分区域记录方式。传统的软驱读写头不够灵敏,所以每条磁道上扇区数都是 18 个,而现在最内圈磁道有 51 个扇区,最外圈磁道则高达 92 个扇区。

以往的软盘盘片材质已不能负担如此高密度的磁道数和扇区数,需要采用一种高密度金属粒涂料来作为存储介质。

3. 磁光盘

磁光盘又称 MO(Magnet Optical)盘,这是一种采用激光和磁场共同作用的磁光方式存储技术。磁光盘既具硬盘的大容量和可读写功能,又有软盘的便携特性,同时具有光盘防磁、抗湿和可靠的特征,因而受到业界注目。磁光盘的记录层很薄,采用对温度极为敏感的磁性材料制成,这些磁性材料在高温下可以被磁化。

磁光盘从 1989 年开始投入使用,是传统的磁盘技术与现代的光学技术结合的产物。磁光盘的外形与传统软盘差不多,但容量比软盘大得多,3.5in 的磁光盘容量可以达到 1.3GB,可重复读写一千万次以上,携带方便,且保存寿命长达 50 年以上,因此在存储图形、图像文件、大型数据库文件方面起着重要的作用。但磁光盘的致命缺点是不能用普通光驱读出。

磁光盘所用的磁层中存在着许多已磁化的磁畴,磁畴的磁化方向与介质表面垂直。初始时,在外界磁场的作用下,全部磁畴转向同一方向。当数据写入时,利用凸透镜进行聚焦,

将高功率激光照射在MO盘记录层上形成极小的光点，当此点的温度上升到约300℃(居里点)时，磁畴随外磁场的作用而改变其原磁化方向。激光迅速移去后，磁畴温度恢复正常，数据被保存在MO盘上。

所谓居里温度，是指材料可以在铁磁体和顺磁体之间改变的温度。当温度低于居里温度时，该物质称为铁磁体，此时材料的磁场很难改变；当温度高于居里温度时，该物质称为顺磁体，这时材料的磁场很容易随周围磁场的改变而改变。

数据的读取是利用低功率的激光探测盘片表面，通过分析反射回来的偏振光的偏振面方向是顺时针还是逆时针，来决定读取的数据是“1”还是“0”。

要进行数据重写时，必须经过“擦”和“写”两步：首先利用中功率激光照射拟擦除的位置，使磁畴翻转恢复到原来的方向，即通过写入“0”来抹去原存数据；然后再根据要求用高功率激光在需要的位置写入数据“1”，这样就完成了数据的重写。

4. 磁带

磁带的位价格要比磁盘便宜很多，整体容量也大一些。磁带是顺序访问的，用户要找一个文件，必须从磁带头开始，而且不能单独修改或移动磁带上的单个文件，必须将整盒磁带的内容删除，然后再全部重写。因此，磁带比较适合作为整个硬盘程序和数据的备份存储器，即大容量的备份存储。

计算机上要备份的数据、要存储的档案可能需要大量空间，一些用户每星期甚至每天都需要备份他们的数据，即将这些数据转移到别的存储介质上，以便为机器留出更多的磁盘空间。

备份整个硬盘数据或修改数据的传统方法是使用磁带，如果磁带容量足够大，用磁带备份整个硬盘的数据是最简单、最有效的方法。在机器上装一个用于备份的磁带机，在机器里插入一卷磁带，选择要备份的驱动器和文件，然后开始备份，备份软件就开始将要备份的数据往磁带上复制，而用户就可以干别的事去了。以后要修改磁带上的部分或全部数据时，将这盒磁带插入磁带机，启动备份程序，选择需要重新存入的文件，剩下的工作就由磁带机来做了。

自动加载磁带机是一个位于单机中的磁带驱动器和自动磁带更换装置，它可以从装有多盘磁带的磁带匣中拾取磁带并放入驱动器中，或者执行相反的过程。自动加载磁带机能够支持例行备份过程，自动为每日的备份工作装载新的磁带。一个拥有工作组服务器的小公司可以使用自动加载机来自动完成备份工作。

磁带库是像自动加载磁带机一样基于磁带的备份系统，磁带库由多个驱动器、多个槽、机械手臂组成，并可由机械手臂自动实现磁带的拆卸和装填。它能够提供同样的基本自动备份和数据恢复功能，但同时具有更先进的技术特点。它可以多个驱动器并行工作，也可以几个驱动器指向不同的服务器来做备份，存储容量达到PB级($1P=2^{50}$)，可实现连续备份、自动搜索磁带等功能，并可在管理软件的支持下实现智能恢复、实时监控和统计，是集中式网络数据备份的主要设备。

8.6.2 基于电子器件的存储器

基于电子器件的闪存卡、固态硬盘和U盘与磁盘、光盘等传统存储产品相比表现出更

为旺盛的生命力。

1. 闪存卡

闪存卡是利用闪存(Flash Memory)技术存储信息的存储器,它是数码相机的最好搭档,所以也被称为数字“胶卷”,和普通的胶卷不同,它可以被擦除,然后可重新使用。对于闪存卡来说,最重要的指标是容量,其次是读写速度。写入速度高意味着数码相机可以迅速地把拍摄的数据传送到闪存卡中,准备好进行下一次拍摄。读出速度高的闪存卡可以缩短图像数据上传到计算机所需的时间。

闪存卡是相当特殊的存储介质,从接口规范和使用来看,它就像一块外置硬盘,但在内部,半导体存储器的特性相当突出。根据不同的生产厂商和不同的应用,闪存卡有CF卡、SM卡、记忆棒、MMC卡、SD卡、xD卡和微硬盘7类。这些闪存卡虽然外观、规格不同,但是技术原理都是相同的。

(1) CF卡

CF(Compact Flash)卡曾经是闪存阵营中的“老大”,CF卡上内置了ATA/IDE控制器,具备即插即用功能,可以兼容绝大部分操作系统。CF是最老也是最成功的闪存标准之一,尤其适合专业相机市场。

(2) SM卡

SM(SmartMedia)卡又称为固态软盘卡(SSFDC),大小与CF卡相似,与CF卡不同之处在于没有内置控制器,控制器集成在数码产品中,目前的数码产品已很少采用SM卡。

(3) 记忆棒

从外形上看,标准的记忆棒比一块口香糖略小,它采用排列在单侧的10针接口与驱动器连接。

(4) MMC卡

MMC卡在SM卡基础上诞生又很快地替代了SM卡。MMC也把控制器一同做到卡上,智能的控制器使得MMC保证兼容性和灵活性。

(5) SD卡

在MMC卡基础上研发的SD卡与MMC具有一定的兼容性,但SD卡的容量要大得多,且读写速度也比MMC卡快4倍。SD卡的衍生产品有Mini SD卡和Micro SD卡,2005年推出Micro SD卡只有指甲般大小,智能手机上使用的扩展内存卡就是这种极小的存储卡。

(6) xD卡

xD卡不仅满足了现有数码相机用户对大存储容量及良好兼容性的需求,而且其袖珍的体积也为生产设计更精致小巧的数码相机打下了基础。在读写兼容性上,xD卡不仅拥有PC卡适配器和USB读卡器,非常容易与个人计算机连接,而且小巧的体积还让它可以插入CF适配器,在使用CF卡的数码相机中使用。

(7) 微硬盘

微硬盘是一款超级迷你硬盘机产品,其最初的容量为340MB和512MB,而现在的产品容量有8GB、16GB和30GB等。可以使用CF卡的大多数设备大都可以直接使用微硬盘。

2. 固态硬盘

固态硬盘(Solid State Disk)是用固态电子存储芯片阵列而制成的硬盘,由控制单元和存储单元(Flash 芯片或 DRAM 芯片)组成。目前,固态硬盘主要用来在便携式计算机中代替传统硬盘。虽然在固态硬盘中已经没有可以旋转的盘状结构,但是依照人们的命名习惯,仍然将其称为"硬盘"。固态硬盘的接口规范和定义、功能及使用方法与普通硬盘的相同,在产品外形和尺寸上也与普通硬盘一致,新一代的固态硬盘普遍采用 SATA-2 接口及 SATA-3 接口。

基于闪存(Flash 芯片)的固态硬盘是固态硬盘的主要类别,其内部构造十分简单,固态硬盘内主体其实就是一块 PCB 板,而这块 PCB 板上最基本的配件就是控制芯片、缓存芯片和用于存储数据的闪存芯片。

固态硬盘的特点如下:

① 读写速度快。采用闪存作为存储介质,读取速度相对机械硬盘更快。最常见的 7200RPM 传统硬盘的寻道时间一般为 12～14ms,而固态硬盘不用磁头,寻道时间几乎为 0。

② 低功耗、无噪音、抗震动、低热量。基于闪存的固态硬盘在工作状态下能耗和发热量较低;没有机械马达和风扇,工作时噪音值为 0dB;内部不存在任何机械活动部件,不怕碰撞、冲击、振动;而且重量轻,工作温度范围大。

③ 寿命限制。固态硬盘的闪存具有擦写次数限制。

目前制约固态硬盘普及的三大问题是:成本、写入次数以及损坏时的不可挽救性。第一,固态硬盘的位价格相对传统硬盘要高出数倍;第二,由于闪存有一定的写入次数限制,寿命结束后会无法写入,变成只读状态;第三,固态硬盘数据损坏后是难以修复的,现时的数据修复技术不可能在损坏的芯片中救回数据,相反传统硬盘或许还能通过一些数据恢复技术挽回一些数据。

3. U 盘

U 盘全称是 USB 闪存驱动器,英文名"USB flash disk"。它是一种基于闪存和 USB 接口的、无须物理驱动器的微型大容量移动存储产品,通过 USB 接口与计算机连接,实现即插即用。

U 盘最大的优点是:小巧便于携带、存储容量大、价格便宜、性能可靠。U 盘中无任何机械式装置,抗震性能极强。另外,还具有防潮防磁、耐高、低温等特性。U 盘可长期保存数据,擦写次数达百万次以上。许多 U 盘还具有写保护功能,可以防止病毒写入 U 盘。

U 盘的主要目的是用来存储数据资料,但目前已超越此目的开发出了更多的功能,例如加密 U 盘、启动 U 盘、杀毒 U 盘、测温 U 盘和音乐 U 盘等。

① 加密 U 盘:加密 U 盘分为两类,一类采用硬件加密技术,通过 U 盘的主控芯片进行加密,安全级别高,不容易被破解,但成本较高;另一类采用软件加密技术,通过外置服务端或内置软件操作,对 U 盘文件进行加密,成本相对较低。

② 启动 U 盘:启动 U 盘也分为两类,一类专门用来作系统启动用的功能性 U 盘,当计算机不能正常开启时进入系统进行相关操作,功能比较单一。另一类是用来维护计算机而

专门制作的强大的功能性U盘,除了可以启动计算机外还可以有磁盘分区、系统杀毒、系统修复、文件备份、密码修改等功能。

③ 杀毒U盘:这是一种将各种杀毒软件U盘版嵌入U盘中,使杀毒软件使用更方便快捷、安全、操作简单,与USB接口相连后即会被主机识别,并不需要烦琐的安装。

④ 测温U盘:测温U盘可分为两类,一类是在计算机上安装了一个软件,通过U盘中的一个测试温度软件,感应出U盘所获取的温度。另一类是直接将测试温度的硬件封装在U盘内,并直接显示在U盘输出LED屏上。

⑤ 音乐U盘:音乐U盘既有U盘的全部存储功能,同时还具备音乐文件的播放功能。一般的音乐U盘外观和普通U盘并无异样,不同之处在于其内置了电池,并多出一个插孔,用来接入配备的耳机,插进去后即可听取MP3、WMA等常见格式音乐,支持上、下曲播放选取,可设置随机播放功能。

8.7 键盘输入设备

键盘是计算机系统不可缺少的输入设备,人们通过键盘上的按键直接向计算机输入各种数据、命令及指令,从而使计算机完成不同的运算及控制任务。

8.7.1 键开关与键盘类型

键盘上的每个按键各起一个开关的作用,故又称为键开关。键开关分为接触式和非接触式两大类。

接触式键开关中有一对触点,最常见的接触式键开关是机械式键,它是靠按键的机械动作来控制开关开启的。当键帽被按下时,两个触点被接通;当键帽被释放时,弹簧恢复原来触点断开的状态。这种键开关结构简单,成本低,但开关通断会产生触点抖动,而且使用寿命较短。

非接触式键开关的特点是:开关内部没有机械接触,只是利用按键动作改变某些参数或者利用某些效应来实现电路的通、断转换。非接触式键开关主要有电容式键和霍尔键两种,其中电容式键是比较常用的。这种键开关无机械磨损,不存在触点抖动现象,性能稳定,寿命长,已成为当前键盘的主流。

按照键码的识别方法,键盘可分为两大类型:编码键盘和非编码键盘。

编码键盘是用硬件电路来识别按键代码的键盘,当某键按下后,相应电路即给出一组编码信息(如ASCII码)送主机去进行识别及处理。编码键盘的响应速度快,但它以复杂的硬件结构为代价,并且其硬件的复杂程度随着键数的增加而增加。

非编码键盘是用较为简单的硬件和专门的键盘扫描程序来识别按键的位置,即当按某键以后并不给出相应的ASCII码,而提供与按下键相对应的中间代码,然后再把中间代码转换成对应的ASCII码。非编码键盘的响应速度不如编码键盘快,但是它通过软件编程可为键盘中某些键的重新定义提供更大的灵活性,因此得到广泛的使用。

8.7.2 键盘扫描

在大多数键盘中,键开关被排列成M行×N列的矩阵结构,每个键开关位于行和列的

交叉处。非编码键盘常用的键盘扫描方法有逐行扫描法和行列扫描法。

1. 逐行扫描法

图 8-11 是采用逐行扫描识别键码的 8×8 键盘矩阵，8 位输出端口和 8 位输入端口都在键盘接口电路中，其中输出端口的 8 条输出线接键盘矩阵的行线（$X_0 \sim X_7$），输入端口的 8 条输入线接键盘矩阵的列线（$Y_0 \sim Y_7$）。通过执行键盘扫描程序对键盘矩阵进行扫描，以识别被按键的行、列位置。

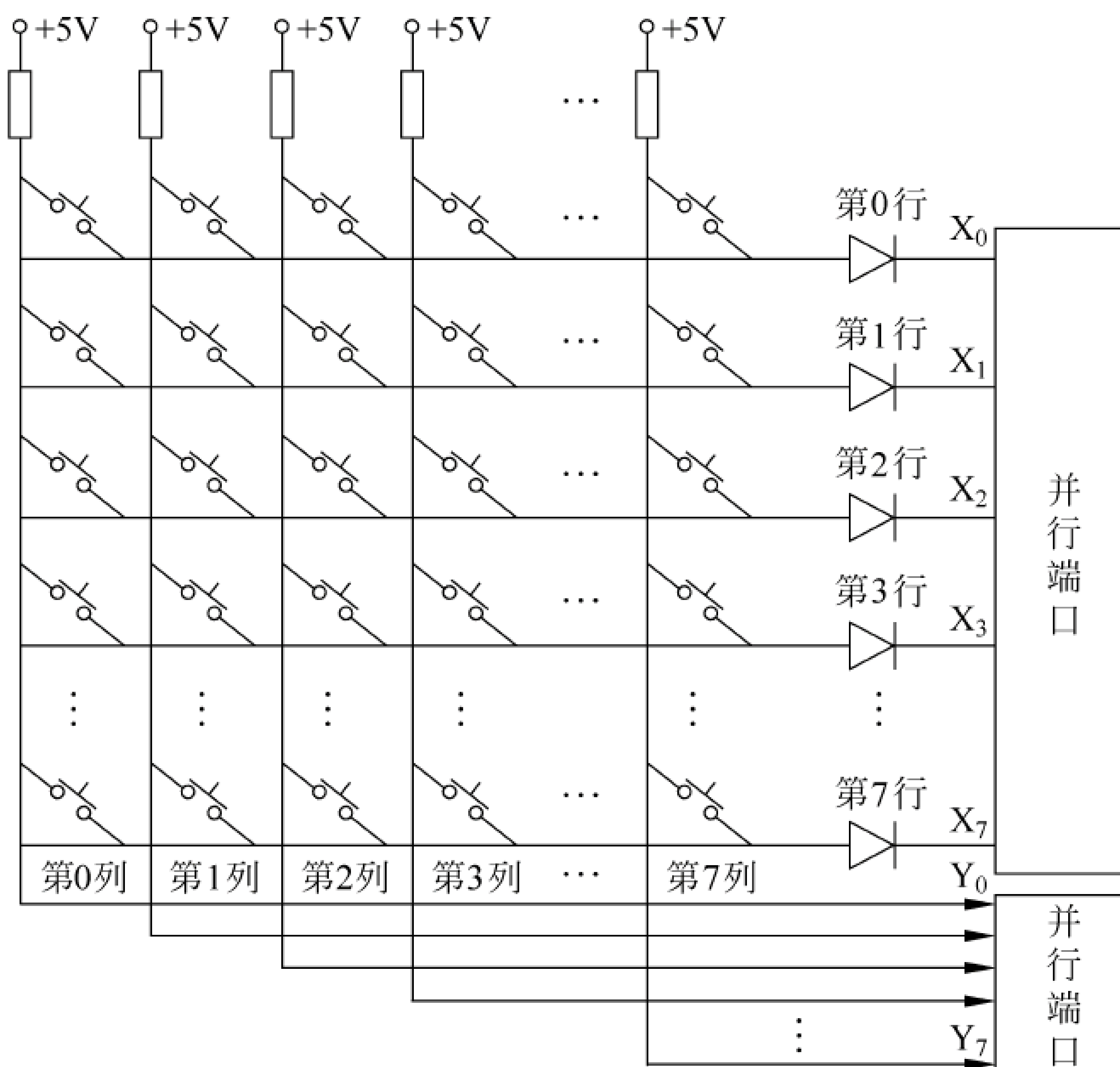


图 8-11　采用逐行扫描法的 8×8 键盘矩阵

键盘扫描程序处理的步骤如下：

① 查询是否有键按下。首先由 CPU 对输出端口的各位置“0”，即使各行全部接地，然后 CPU 再从输入端口读入数据。若读入的数据全为“1”，表示无键按下；只要读入的数据中有一个不为“1”，表示有键按下。接着要查出按键的位置。

② 查询已按下键的位置。CPU 首先使 $X_0=0$，$X_1 \sim X_7$ 全为“1”，读入 $Y_0 \sim Y_7$，若全为“1”，表示按键不在这一行；接着使 $X_1=0$，其余各位为全“1”，读入 $Y_0 \sim Y_7$，……直至 $Y_0 \sim Y_7$ 不全为“1”为止，从而确定了当前按下的键在键盘矩阵中的位置。

③ 按行号和列号求键的位置码。得到的行号和列号表示按下键的位置码。

对于接触式键开关，为避免触点抖动造成的干扰，通常采用软件延时的方法来等候信号稳定。具体的做法是：在检查到有键按下以后延时一段时间（约 20ms），再检查一次是否有键按下。若这一次检查不到，则说明前一次检查结果为干扰或者抖动；若这一次检查到有键按下，则可确认这是一次有效的按键。

2. 行列扫描法

在扫描每一行时，读列线，若读得的结果为全“1”，说明没有键按下，即尚未扫描到闭合

键;若某一列为低电平,说明有键按下,而且行号和列号已经确定。然后用同样的方法,依次向列线扫描输出,读行线。如果两次所得到的行号和列号分别相同,则键码确定无疑,即得到闭合键的行列扫描码。

8.7.3 微型计算机键盘

从按键的数量上看,微型计算机的键盘有83键(PC/XT)、84键(PC/AT)、101和102键(386、486机)、104键(Pentium)、105键、108键、109键等多种。

键盘通常通过设在主板上的键盘接口连到主机上,人们通过键盘输入的数据是在主机的BIOS程序的控制下传送到主机的CPU中进行处理的。图8-12为PC/XT键盘与接口框图,图中虚线的左侧部分是PC/XT键盘,右侧部分是键盘接口,位于微机主板上。

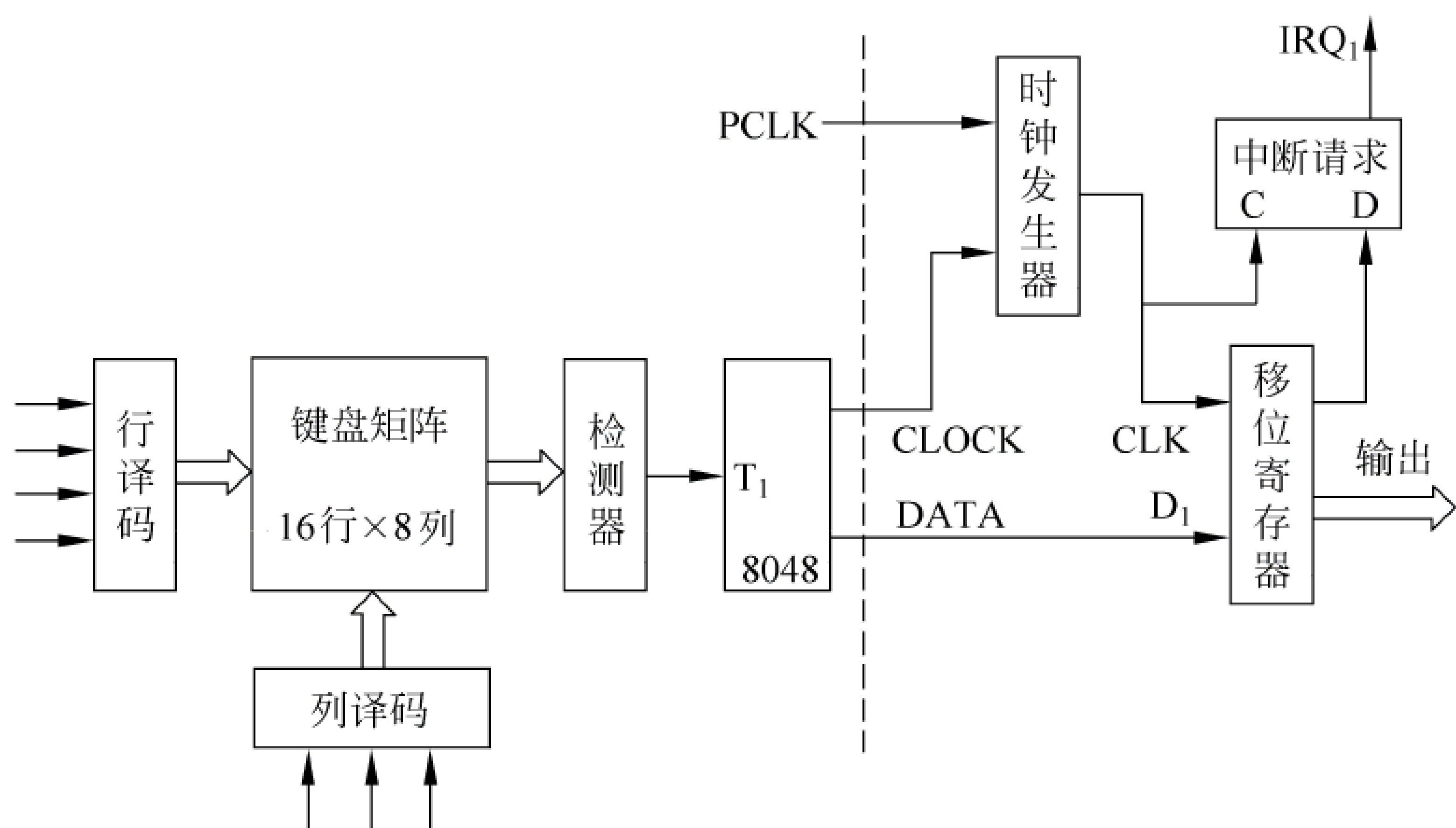


图8-12 PC/XT键盘与接口

1. 键盘控制电路

PC系列键盘一般由键盘矩阵和以单片机或专用控制器为核心的键盘控制电路组成,被称为智能键盘。单片机通过执行固化在ROM中的键盘管理和扫描程序,对键盘矩阵进行扫描,发现、识别按下键的位置,形成与按键位置对应的扫描码,并以串行的方式送给微机主板上的键盘接口电路,供系统使用。

PC/XT键盘(83键)采用16行×8列矩阵结构,由8048单片机实现闭合键检测、键码识别和与主机通信的控制。8048通过译码器,分别产生16个行扫描信号和8个列扫描信号。扫描方式采用行列扫描法,即先逐列为“1”地进行列扫描,矩阵检测器输出送8048测试端T_1,可判断是否有行线输出“1”,从而得到闭合键的列号。然后采用同样的方法,逐行为“1”地进行行扫描,得到闭合键的行号。8048将列号和行号拼成一个7位的扫描码(列号为前3位,行号为后4位),例如第4列第7行键被按下,则得到闭合键的扫描码为47H。

在8048中有一个20字节的缓冲队列,能暂存20个扫描码。当多键滚按时,若干按键的扫描码便被放入缓冲队列。按先进先出的原则从缓冲区取出扫描码送往接口,以免高速按键时主机来不及进行中断响应和处理。8048的键盘扫描程序还能完成去抖动、延时自动拍发等复杂功能。

键盘内部的单片机根据按键位置向主机发送的仅是该按键位置的键扫描码。当键按下时，输出的数据称为接通扫描码；当键松开时，输出的数据称为断开扫描码。

对于83键键盘，由键盘扫描电路得到的接通扫描码与键号（键的位置编号）相等，用1个字节表示，断开扫描码也用1个字节表示，其值为接通扫描码加80H。例如，“A”的键号为30，接通扫描码为1EH，断开扫描码为9EH。

对于84/101/102/104扩展键盘，由于键位置发生变化，其接通扫描码与键号不相等。但是，接通扫描码仍用1个字节表示，断开扫描码用两个字节表示，其值为接通扫描码前加1个字节的前缀F0H。仍以“A”键为例，它的键号为31，接通扫描码为1CH，断开扫描码为F0H、1CH。

2. 键盘接口电路

键盘接口电路一般在微机主板上，通过电缆与键盘连接，串行地接收键盘送来的扫描码，或者向键盘发送命令，要求键盘完成一定的工作（如自检）。其功能主要有：

① 串行接收键盘送来的接通扫描码和断开扫描码，转换成并行数据并暂存。

② 收到一个完整的扫描码后，立即向主机发中断请求。

③ 主机中断响应后读取扫描码，并转换成相应的ASCII码存入键盘缓冲区。对于控制键，设置相应的状态。

④ 接收主机发来的命令，传送给键盘，并等候键盘的响应，自检时用以判断键盘的正确性。

对于83键键盘，键盘接口电路主要由8255A-5和74LS322移位寄存器构成，称为PC标准键盘接口。对于扩展键盘，键盘接口电路主要由单片机8042/8742构成，称为扩展键盘接口。由于扩展键盘的扫描码与系统扫描码不一致，因此8042/8742除了完成上述功能以外，还要完成由键盘扫描码到系统扫描码的转换。所谓系统扫描码就是与相应83键键盘中同字符的接通扫描码。

从键盘送来的串行扫描码在移位寄存器中由时钟控制依次右移，组装成并行扫描码，然后向主机CPU发出中断请求IRQ_1。主机CPU响应键盘中断请求后，执行由BIOS提供的键盘中断处理程序（09H类型中断）。该程序首先以并行方式从接口取出扫描码，接着对收到的扫描码进行识别，判断按下的键是字符键还是控制键，由中断服务程序通过查表，将扫描码转换为相应的ASCII码或扩充码后送入键盘缓冲区，中断处理完毕返回主程序。当系统或用户需要键盘输入时，可直接在主程序中以软中断指令（INT 16H）的形式调用BIOS的键盘I/O程序，从键盘缓冲区中取走所需的字符。

在微型计算机中，所有字母、数字等由键盘输入后均以ASCII码的形式存放在键盘缓冲区，在存放时，每个键的编码占两个字节，其中高字节仍是系统扫描码，低字节是由中断服务程序转换成的ASCII码。另外，还有一些键没有对应的ASCII码，如命令键、组合功能键，对于这些键则用扩充码表示。扩充码存放时高位字节是扩充码，低位字节是00H。这就是说，BIOS中断服务程序执行时首先检查输入的系统扫描码是否可以转换成ASCII码。如果可以，则转换成ASCII码，存入键盘缓冲区；如果不可以，则转换成扩充码，存入键盘缓冲区。

键盘缓冲区是一个先进先出的循环队列，其容量（16个字）足以满足操作员快速输入键

符的需要。键盘缓冲区是键盘中断程序(09H 类型中断)与键盘 I/O 程序(INT 16H)之间进行数据传递的媒介体,进队列即由 BIOS 中断服务程序将键盘输入的系统扫描码转换成 ASCII 码或扩充码,按“先进先出”的原则输入到键盘缓冲区中;出队列即由主机执行软件中断 INT 16H,按同样的原则读取键盘缓冲区中的 ASCII 码或扩充码予以处理或执行。

8.8 其他输入设备

目前,计算机系统常用的输入设备除键盘外,还有鼠标、扫描仪、光笔、数字化仪等。键盘输入的是字符和数字信息,鼠标主要输入矢量信息和坐标数据,而扫描仪主要输入图形、图像信息。

8.8.1 鼠标器

鼠标器是控制显示器光标移动的输入设备,由于它能在屏幕上实现快速精确的光标定位,可用于屏幕编辑、选择菜单和屏幕作图,鼠标器已成为计算机系统中必不可少的输入设备。

鼠标器按其内部结构的不同可分为机械式、光机式和光电式三大类。尽管结构不同,但从控制光标移动的原理上讲三者基本相同,都是把鼠标器的移动距离和方向变为脉冲信号送给计算机,计算机再把脉冲信号转换成显示器光标的坐标数据,从而达到指示位置的目的。

1. 机械式鼠标

机械鼠标的结构最为简单,由鼠标底部的胶质小球带动 X 方向滚轴和 Y 方向滚轴,在滚轴的末端有译码轮,译码轮附有金属导电片与电刷直接接触。鼠标的移动带动小球的滚动,再通过摩擦作用使两个滚轴带动译码轮旋转,接触译码轮的电刷随即产生与二维空间位移相关的脉冲信号。目前,纯粹的机械鼠标已经基本消失,我们见到的底部带小球的鼠标都是光机式鼠标。

2. 光机式鼠标

光机鼠标顾名思义就是一种光电和机械相结合的鼠标,在机械鼠标的基础上,将磨损最厉害的接触式电刷和译码轮改为非接触式的 LED 对射光路元件。当小球滚动时,X、Y 方向的滚轴带动码盘旋转。安装在码盘两侧有两组发光二极管和光敏三极管,LED 发出的光束有时照射到光敏三极管上,有时则被阻断,从而产生了两组相位相差 90°的脉冲序列。脉冲的个数代表鼠标的位移量,而相位表示鼠标运动的方向。由于采用的是非接触部件,使磨损率下降,从而提高了鼠标的寿命,也能在一定范围内提高鼠标的精度。

3. 光电式鼠标

光电鼠标内部有一个发光二极管,通过其发出的光线,照亮光电鼠标底部表面,然后将反射回来的一部分光线,经过一组光学透镜,传输到一个光感应器件内成像。这样,当光电鼠标移动时,其移动轨迹便会被记录为一组高速拍摄的连贯图像。最后利用光电鼠标内部

的一块专用图像分析芯片对移动轨迹上摄取的一系列图像进行分析处理，通过对这些图像上特征点位置的变化进行分析，来判断鼠标的移动方向和移动距离，从而完成光标的定位。

除前述3类传统鼠标外，现在还出现了激光鼠标。

激光鼠标其实也是光电鼠标，只不过是用激光代替了普通的LED光。相对于传统的鼠标来说，激光鼠标是将鼠标本身的激光照射在物体表面，同时物体表面的激光反射回激光鼠标的传感器之内。由于激光能对图像产生更大的反差，所以激光鼠标相对于传统鼠标来讲，灵敏度更佳，定位更加准确，扫描速度更快。

鼠标与计算机的联接方式分为有线和无线两种，有线鼠标按接口类型可分为串行鼠标、PS/2鼠标、USB鼠标几种。串行鼠标是通过串行口与计算机相连，有9针接口、25针接口两种。PS/2鼠标通过一个6针微型DIN接口与计算机相连，它与键盘的接口非常相似，使用时应注意区分。USB鼠标通过一个USB接口，直接插在计算机的USB口上。

无线鼠标主要有以下几种：

① 27MHz的无线鼠标，其发射距离在2m左右，而且信号不稳定，相对比较低档。

② 2.4GHz的无线鼠标，其接受信号的距离在7～15m，信号比较稳定，目前市场主要是这一种。

③ 蓝牙鼠标，其发射频率和2.4GHz一样，接受信号的距离也一样。蓝牙有一个最大的特点就是通用性，如果计算机带蓝牙功能，那么不需要蓝牙适配器就可以直接连接，可以节约一个USB插口。

鼠标按键数可以分为双键、三键和多键鼠标。三键鼠标的中键在某些特殊程序中往往能起到事半功倍的作用；多键鼠标是新一代多功能鼠标，如有的鼠标上带有滚轮，使得上下翻页变得极其方便。有的新型鼠标上除了有滚轮，还增加了拇指键等快捷按键，进一步简化了操作程序。

8.8.2 其他定位设备

随着便携式计算机的出现，鼠标器已不能适应新的要求，因此又出现了一些新的定位设备。

1. 轨迹球

轨迹球的结构颇像一个倒置的鼠标，好像在小圆盘上镶嵌一颗圆球。轨迹球的功能与鼠标相似，朝着指定的方向转动小球，光标就在屏幕上朝着相应的方向移动。轨迹球可以独立使用，也常常嵌在键盘上，其优点是不像鼠标那样必须有可供滑动的较大的空间。

2. 跟踪点

跟踪点是一个压敏装置，只有铅笔上的橡皮大小，所以可嵌在按键之间，用手指轻轻推它，光标就朝着指点的方向移动。

3. 触摸板

触摸板是一种方便的输入设备，它的表面对压力和运动敏感，当用手指轻轻在触摸板滑动时，屏幕上的光标就同步运动。有的触摸板周围设有按钮，其作用与鼠标的按钮相同，另

一些触摸板,则是通过轻敲触摸板表面完成与点击鼠标相同的操作。

8.8.3 扫描仪

扫描仪是一种光、机、电一体化的高科技产品,它是将各种形式的图像信息输入计算机的重要工具,是继键盘和鼠标之后的第三代计算机输入设备,也是功能极强的一种输入设备。从最直接的图片、照片、胶片到各类图纸图形以及文稿资料都可以用扫描仪输入到计算机中,进而实现对这些图像形式信息的处理、管理、使用、存储、输出等。配合文字识别软件,还可以将扫描的文稿转换成计算机的文本形式。

1. 扫描仪的组成部分及工作原理

自然界每一种物体都会吸收特定的光波,而没有吸收的光波就会被反射出去。扫描仪就是利用这种特性来完成对稿件的读取的。扫描仪在工作时会发出强光照射在稿件上,没有被吸收的光线将被反射到光学感应器上。光学感应器接收到这些信号后,再将这些信号传送到数模转换器,数模转换器再将其转换成计算机能够读取的信号,然后通过驱动程序转换成显示器上能看到的正确图像。欲扫描的稿件通常可以分为反射稿和透射稿。反射稿泛指一般的不透明文件,例如报纸、杂志等。透射稿包括幻灯片(正片)或底片(负片)。如果经常需要扫描透射稿,那就必须选择具备光罩(光板)功能的扫描仪。

扫描仪的光学读取装置相当于人的眼球,其重要性不言而喻。目前,扫描仪所使用的光学读取装置有 CCD 和 CIS 两种。

(1) CCD(Charge Coupled Device)

CCD 的中文名称为电荷耦合装置。它采用电荷耦合的微型半导体感光芯片作为扫描仪的核心。CCD 与日常使用的半导体集成电路相似,在一片硅单晶上集成了几千到几万个光电三极管,这些光电三极管分为 3 列,分别用红、绿、蓝色的滤色镜罩住,从而实现彩色扫描。光电三极管在受到光线照射时可以产生电流,经放大后输出。采用 CCD 的扫描仪技术经过多年的发展已经相当成熟,是目前市场上主流扫描仪主要采用的感光元件。CCD 的优势主要在于:扫描的图像质量近年来提高很大;具有一定的景深,能够扫描凹凸不平的物体;温度系数比较低,对周围环境温度的变化可以忽略不计。CCD 的缺陷主要有:由于数千个光电三极管的距离很近(微米级),在各光电三极管之间存在着明显的漏电现象,各感光单元的信号产生干扰,降低了扫描仪的实际清晰度;由于采用了反射镜、透镜,会产生图像色彩偏差和像差,需要通过软件进行校正;由于 CCD 需要一套精密的光学系统,故扫描仪体积不可能做得很小。

(2) CIS(Contact Image Sensor)

CIS 的中文名称为接触式图像感应装置。它采用一种触点式图像感光元件(光敏传感器)来进行感光,在扫描平台下 1~2mm 处,300~600 个红、蓝、绿三色 LED(发光二极管)传感器紧密排列在一起产生白色光源,取代了 CCD 扫描仪中的 CCD 阵列、透镜、荧光管和冷阴极射线管等复杂结构,把 CCD 扫描仪的光、机、电一体变为 CIS 扫描仪的机、电一体。但 CIS 技术也有不足之处:CIS 固有的感光特性决定了这种扫描仪需要一次扫描、三次曝光,所以扫描速度比较慢;由于 CIS 没有景深的概念,原稿必须与感光元件靠得很近才行,这样无法进行实物扫描;而且目前 CIS 感光元件的性能决定了 CIS 扫描仪分辨率不高,加上 CIS

光源的均匀性不够好,使得 CIS 扫描仪的扫描图像质量和色彩真实度不是太好,甚至比不上一些低价位的 CCD 扫描仪。但是,这类扫描仪具有体积小、重量轻、器件少和抗震性较高的优点,而且生产成本很低。

2. 扫描仪的主要性能指标

(1) 分辨率

分辨率通常是指图像每英寸中有多少个像素(Pixel)。分辨率对图像的质量有很大的影响,通常分辨率越高,扫描输入的时间就越长。扫描仪的分辨率又可细分为光学分辨率和最大分辨率两种。

① 光学分辨率。光学分辨率是扫描仪最重要的性能指标之一,它直接决定了扫描仪扫描图像的清晰程度。扫描仪的光学分辨率用每英寸长度上的点数,即 DPI 来表示。通常,低档扫描仪的光学分辨率为 300×600DPI,中高档扫描仪的光学分辨率为600×1200DPI。

光学分辨率指的是扫描仪实际工作时的分辨能力,也就是在每英寸上它所能扫描的光学点数。通常这个数值是不变的,因为它由光学感应元件的性能决定。

② 最大分辨率。最大分辨率又称软件分辨率,通常是指利用软件插值补点的技术模拟出来的分辨率。光学分辨率为 300×600DPI 的扫描仪一般最大分辨率可达 4800DPI,而 600×1200DPI 的扫描仪则更高达 9600DPI。这实际上是通过软件在真实的像素点之间插入经过计算得出的额外像素,从而获得的插值分辨率。插值分辨率对于图像精度的提高并无好处,事实上只要软件支持,而用户的机器又强劲,这种分辨率完全可以做到无限大。

(2) 色彩深度值

色彩深度值又称为色阶或色彩位数,指的是扫描仪色彩识别能力的大小。扫描仪是利用 R(红)、G(绿)、B(蓝)三原色来读取数据的,如果每个原色以 8 位数据来表示,总共就有 24 位,即扫描仪有 24 位色阶;如果每个原色以 12 位数据来表示,总共就有 36 位,即扫描仪有 36 位色阶,它所能表现出的色彩将会有 680 亿(2^{36})色以上。较高的色彩深度位数可以保证扫描仪反映的图像色彩与实物的真实色彩尽可能的一致,而且图像色彩会更加丰富。一般光学分辨率为 300×600DPI 的扫描仪其色彩深度为 24 位、30 位,而 600×1200DPI 的为 36 位,最高的为 48 位。

(3) 灰度值

灰度值是指进行灰度扫描时对图像由纯黑到纯白整个色彩区域进行划分的级数,又称为灰度动态范围。灰度值越高,扫描仪能够表现的暗部层次就越细。灰度值的大小对于扫描仪正负片通常会有较大的影响。编辑图像时一般都使用到 8 位,即 256 级,而主流扫描仪通常为 10 位,最高可达 12 位。

8.9 打印输出设备

打印机是计算机系统的主要输出设备之一,打印机的功能是将计算机的处理结果以字符或图形的形式印刷到纸上,转换为书面信息,便于人们阅读和保存。由于打印输出结果能够永久性保留,故称为硬拷贝输出设备。

8.9.1 打印机概述

按照打印的工作原理不同,打印机分为击打式和非击打式打印机两大类。击打式打印机是利用机械作用使印字机构与色带和纸相撞击而打印字符的,它的工作速度不可能很高,而且不可避免地要产生工作噪声,但是设备成本低,针式打印机就是使用最广泛的击打式打印机。非击打式打印机是采用电、磁、光、喷墨等物理或化学方法印刷出文字和图形的,由于印字过程没有击打动作,因此印字速度快、噪声低,但一般不能复制多份,目前主要有喷墨打印机、激光打印机等。

打印机按照输出工作方式可分为串式打印机、行式打印机和页式打印机3种。串式打印机是单字锤的逐字打印,在打印一行字符时,不论所打印的字符是相同或不同的,均按顺序沿字行方向依次逐个字符打印,因此打印速度较慢,一般用字每秒(CPS)来衡量其打印速度。行式打印机是多字锤的逐行打印,一次能同时打印一行(多个字符),打印速度较快,常用行每分(LPM)来衡量其速度。页式打印机一次可以输出一页,打印速度最快,一般用页每分(PPM)来衡量其速度。

打印机按印字机构不同,可分为固定字模(活字)式打印和点阵式打印两种。固定字模式打印机是将各种字符塑压或刻制在印字机构的表面上,印字机构如同印章一样,可将其上的字符在打印纸上印出;而点阵式打印机则借助于若干点阵来构成字符。固定字模式打印的字迹清晰,但字模数量有限,组字不灵活,不能打印汉字和图形,所以基本上已被淘汰。点阵式打印机以点阵图拼出所需字形,不需固定字模,它组字非常灵活,可打印各种字符(包括汉字)和图形、图像等。现在人们普遍有一种误解,只把针式打印机看作点阵打印机,这是不全面的。事实上,非击打式打印机输出的字符和图形也是由点阵构成的。

打印机通常有两种工作模式,即文本模式(字符模式)和图形模式。

1. 文本模式

在这种方式中,主机向打印机输出字符代码(ASCII码)或汉字代码(国标码),打印机则依据代码从位于打印机上的字符库或汉字库中取出点阵数据,在纸上“打”出相应字符或汉字。与图形模式相比,文本模式所需传送的数据量少,占用主机CPU的时间少,因而效率较高,但所能打印的字符或汉字的数量受到字库的限制。

2. 图形模式

在图形模式中,主机向打印机直接输出点阵图形数据,有一个“1”就“打”一个点。在这种模式下,CPU能灵活控制打印机输出任意图形,从而可打印出字符、汉字、图形、图像等。但是,图形模式所需传送的数据量大,占用主机大量的时间。例如,打印一个24×24点阵的汉字,传送字符点阵图形的数据量(72个字节)远大于传送字符代码时的数据量(2个字节)。

8.9.2 打印机的主要性能指标

有关打印机的性能指标主要有:分辨率、打印速度、打印幅面、接口方式和缓冲区的大小等。

1. 分辨率

打印机的打印质量是指打印出的字符的清晰度和美观程度，用打印分辨率表示，单位为每英寸打印多少个点(DPI)。大多数打印机的分辨率在垂直和水平方向上是相同的，目前激光打印机的分辨率为600DPI，甚至可达1200DPI。至于精密照排机，低档的在700～2000DPI，高档的则可达2000～3000DPI。

2. 打印速度和打印幅面

不同类型的打印机具有不同的打印速度，每种类型又有高、中、低速之分。

打印机的打印幅面有许多种，一般家庭用户使用A4幅面的就可以了。

3. 接口方式

打印机的接口可以是标准配置并行接口，也可以是USB接口。

4. 缓冲区

最简单的缓冲区只能存放一行打印信息，当这一行信息打印完后，即清除掉缓冲区的信息，并告诉主机"缓冲区空"，主机将再发送新的信息给打印机，如此反复，直到所有信息打印完毕为止。在CPU不断升级的情况下，为了解决计算机和打印机速度的差异，必须扩大打印机的缓冲区。缓冲区越大，一次输入数据就越多，打印机处理打印所需的时间就越长，因此，与主机的通信次数就可以减少，使主机的效率提高。

8.9.3 针式打印机工作原理

针式打印机在打印机历史上曾经占有重要的地位，其价格便宜，耐用，可以打印多层纸，但它较低的打印质量和打印速度以及很大的工作噪声也使它无法适应高质量、高速度的打印需要，所以在普通家庭及办公应用中逐渐被喷墨和激光打印机所取代，只在需要使用复写打印的场合中使用。

针式打印机是由若干根打印针印出$m\times n$个点阵组成的字符或汉字、图形。这里m表示打印的列数，n表示打印的行数。点阵越密，印字的质量就越高。需要注意的是，字符由$m\times n$个点阵组成，并不意味着打印头就装有$m\times n$根打印针。串式针打的打印头上一般只装有一列n根打印针(也有的分为两列)，通常所讲的9针、24针打印机指的就是打印头上打印针的数目。打印头是打印机的关键部件，打印机的打印速度、打印质量和可靠性在很大程度上取决于打印头的功能和质量。

在9针打印机中，将9根打印针排成纵向一列，每次打印一列，印完一列后打印头沿水平方向向右移动一步，m步之后，形成一个$m\times n$点阵的字形。在24针打印机中，一般交错排成两列，每列12根针，分别称为奇数号针和偶数号针。打印时，打印头从左到右打印，一列的24个点是分两次打印出来的。由于点的纵向间距非常小，甚至能相互覆盖一部分，所形成的图形轮廓连贯光滑，印字质量较9针打印机高。

打印头装在一个小车(称为字车)上，由步进电机驱动，可进行水平移动与精确定位。打印头里的钢针在驱动电路的控制下，打击色带和纸，从而形成一行字符。在打印一行字符的

过程中,打印纸不动。在打印完一行后,输纸机构带动打印纸向前推进一行,而色带传动机构也将色带转动一定尺寸,使打击次数均匀地分布在整盘色带上。针式打印机可以通过调整打印头与纸张的间距来适应打印纸的不同厚度,而且可以改变打印针的力度以调节打印的清晰度。

针式打印机有单向打印和双向打印两种。若一行字符打印完,在输纸的同时,打印头左移返回到起始位置(回车),重新由左向右打印,这就称为单向打印。而双向打印指的是自左向右一行字符打印完毕后,打印头无须回车,在输纸的同时,打印头再从右向左打印下一行,做反向打印。由于省去了空回车时间,所以双向打印的打印速度较单向打印大大提高。

针式打印机控制电路如图 8-13 所示。主机要输出打印信息时,首先要检查打印机所处的状态。当打印机空闲时,允许主机发送字符。打印机开始接收从主机送来的字符代码(ASCII 码),先判断它们是可打印的字符还是只执行某种控制操作的控制字符(如"回车""换行"等)。如果是可打印的字符,就将其代码送入打印行缓冲区(RAM)中,接口电路产生回答信息,通知主机发送下一个字符。如此重复,把要打印的一行字符的代码都存入数据缓冲区。当缓冲区接收满一行打印的字符后,停止接收,转入打印。

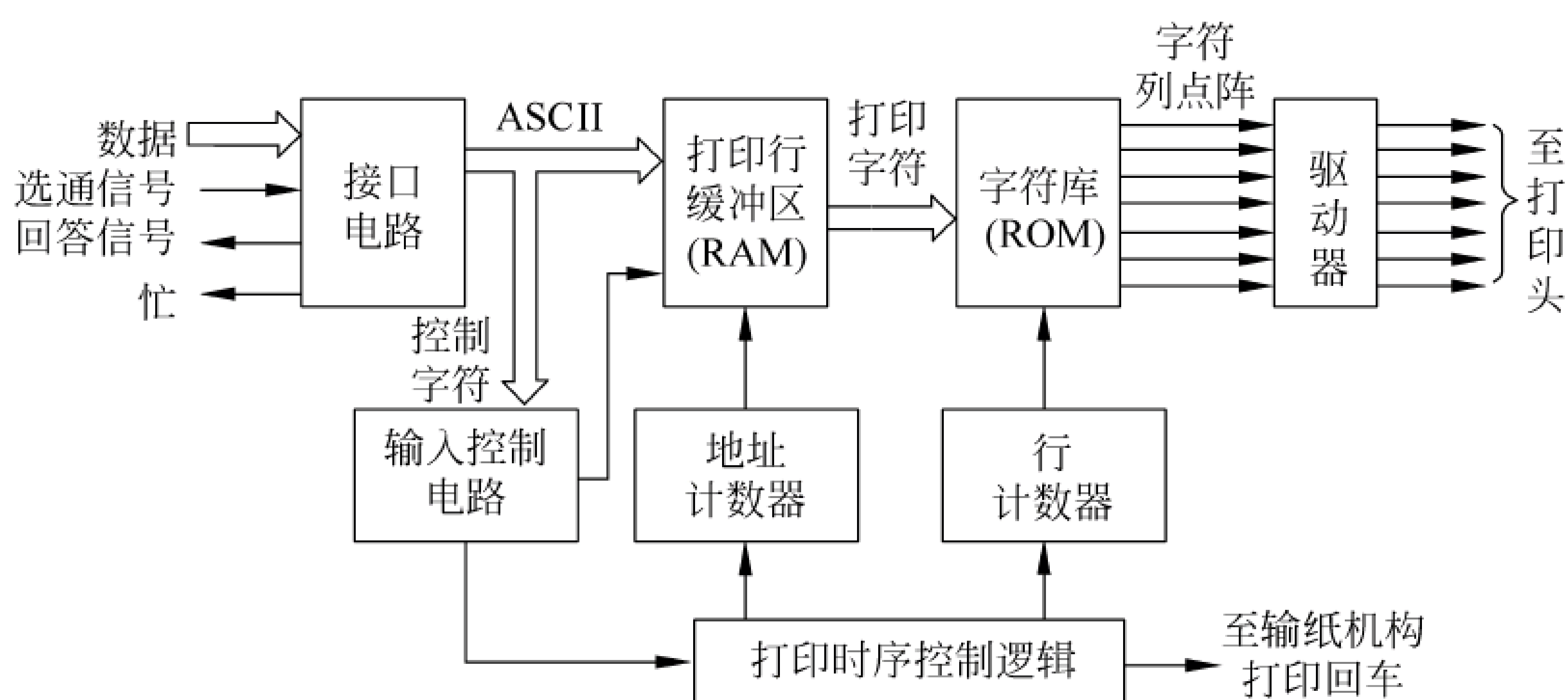


图 8-13 针式打印机控制电路

打印机的字符库中存放着所有字符的列点阵码。打印时,首先形成打印字符的首列点阵的地址,然后按顺序在字符库中一列一列地找出字符的点阵,送往打印头控制驱动电路,激励打印头出针打印。一个字符打印完,字车移动几列,再继续打印下一个字符。一行字符打印完后,请求主机送来第二行打印字符代码,同时输纸机构使打印纸移动一个行距。

一般针式打印机内部只带西文字符库,它只能支持文本的打印。这种打印机若想打印中文,应使打印机处于图形模式。

针式打印机多为单色打印机,现在也出现了彩色针式打印机。针式彩色打印机的结构与单色打印机相同,只是增加了色彩功能控制。彩色打印机利用了三基色混色原理,使用的色带上除了有一条黑色带外,还有红、蓝、黄色 3 条色带,其他的颜色用红、蓝、黄三色混合多次打印组合而成。

彩色打印机的字车上所附的色带不仅能够在水平方向上横向往复运动,而且能够上下移动,这样就可以用一个打印头撞击不同颜色的色带进行彩色打印。为了实现多种彩色的打印,彩色打印机的打印控制电路还增加了色带选择电路及其他附属电路。彩色打印机打印时还是像单色打印机那样,对于每种颜色的色带都是按从左到右的顺序击打,不同的是它

要选择相应的色带。如果打印的是色带上的三基色，则直接选择相应的色带打印即可；如果打印的不是色带上的三基色，则需要利用三基色进行配色，即在同一点上选择不同的色带击打，混合成各种颜色。

8.9.4 喷墨打印机工作原理

喷墨式打印机也属于点阵式打印的一种，它的印字原理是使墨水在压力的作用下，从孔径或狭缝尺寸很小的喷嘴喷出，成为飞行速度很高的墨滴，根据字符点阵的需要，对墨滴进行控制，使其在记录纸上形成文字或图形。喷墨打印机的喷墨方式有两种：连续式和随机式。早期的喷墨打印机以及当前大幅面的喷墨打印机采用连续式喷墨技术，普通喷墨打印机多采用随机式喷墨技术。

1. 连续式喷墨技术

连续式是指连续不断地喷射墨水，首先给墨水加压，使墨水流通过喷嘴连续喷射而粒子化。因为墨水带有正离子，当粒子化的墨水穿过高压电场时，就发生偏转，故可用高压电场控制印字。

当带有正离子的墨水由喷嘴喷出后，墨水束粒子化为小水滴，穿进偏转电极，若想印字，则此时偏转电极上的电压为零，墨水小滴穿过挡板的小孔，喷射在记录纸上。如果不希望印字，就在偏转电极上加±400V的电压，使墨水滴发生偏转，喷射在挡板上，经墨水回收管流入废墨水瓶中。连续式喷墨系统具有频率响应高，可实现高速打印等优点。但是，这种打印机的结构比较复杂，对墨水需要加压装置，终端要有回收装置来回收不参与印字的墨水滴，在墨水循环过程中需要设置过滤器以过滤混入的杂质和气体。采用这种技术的喷墨打印机目前市场上已经极少见到。

2. 随机式喷墨技术

随机式喷墨打印机的墨滴只有在需要打印时才从喷嘴中喷出（又称按需式喷墨打印机），因而不需要过滤器和复杂的墨水循环系统。由于受射流惯性的影响，墨水的喷射速度低于连续式喷墨打印机。为了提高喷射速度，喷头一般由多个喷嘴组成，其结构和排列与针式打印机的打印头相似。随机式喷墨打印机又可分为压电式和气泡式两种。

压电式喷墨打印机的喷头内装有墨水，在喷嘴上下两侧各放置有一块压电陶瓷，利用它在电压作用下会发生形变的原理，适时地把电压加到它的上面，使其变形产生压力，挤压喷头喷出墨滴，在输出介质表面形成图案。用压电喷墨技术制作的喷墨打印头成本比较高，为了降低用户的使用成本，一般都将打印头和墨盒做成分离的结构，更换墨水时不必更换打印头。

气泡式打印机在喷头上设置了加热元件。当脉冲作用于加热元件上，加热元件急速升温，将喷头中的一部分墨汁气化，形成一个具有喷射力量的气泡，并将墨水顶出喷到输出介质表面，形成图案或字符。采用这种技术的打印喷头通常都与墨盒做在一起，更换墨盒时即同时更新打印头。为降低使用成本，在墨盒刚刚用完，可立即加注专用的墨水，只要方法得当，可以节约耗材费用。

通常所说的喷墨打印机是指液态喷墨打印机，它具有整机价格低、工作噪音低、耗电少、

重量轻、输出印字质量接近低档的激光打印机等优点,同时又能实现廉价的真彩色打印。与针式打印机相比,喷墨打印机对墨水的质量要求很高,使得耗材的成本较高,而且墨水大多怕受潮。

除液态喷墨打印机外,还有一种固态喷墨打印机。固态喷墨技术是Tektronix(泰克)公司于1991年推出的专利技术,它所使用的相变墨在室温下可变为固态,打印时墨被加热液化后喷射到介质上,由于此种墨附着性好、色彩鲜亮、耐水性能好,并且不存在打印头因墨水干涸而造成的堵塞问题。但是,采用固态油墨的打印机目前因生产成本比较高,所以产品比较少。

8.9.5 激光打印机工作原理

激光打印机是一种光、机、电一体、高度自动化的计算机输出设备,其成像原理与静电复印机相似,结构比针式打印机和喷墨打印机都复杂得多。它主要由激光器、激光扫描系统、以碳粉与感光鼓为主的碳粉盒、字形发生器、电子照相转印机构和电路部分组成,如图8-14所示。

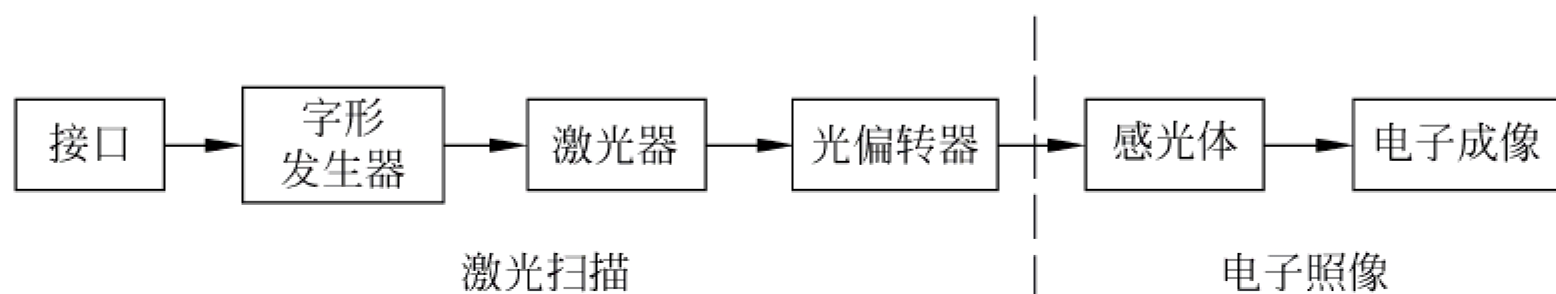


图8-14 激光打印机的组成

感光鼓是激光打印机的核心,这是一个用铝合金制成的圆筒,其表面镀有一层半导体感光材料,通常是硒,所以又常将它称为硒鼓。激光打印机的打印过程中的6个步骤:充电、扫描曝光、显影、转印、定影和清除残像都是围绕感光鼓进行的。

① 充电:对硒鼓进行充电,使其表面均匀地带上正(负)电荷。

② 扫描曝光:扫描曝光也可以叫做“书写”,由控制电路控制激光束对硒鼓表面进行扫描照射,在需印出内容的地方关闭激光束,在不需印出的地方打开激光束。随着带正(负)电荷的感光鼓表面的转动,遇有激光源照射时,鼓表面曝光部分变为良导体,产生光电流,使其失去表面电荷。而未曝光的鼓表面仍保留电荷,从而在硒鼓上形成静电“潜像”。

③ 显影:带有“潜像”的硒鼓表面继续运动,通过碳粉盒时,带电荷的部分吸附上碳粉,从而在鼓面上显影成可见的字符碳粉图像。

④ 转印:显影的表面同打印纸接触时,在外电场的作用下,碳粉被吸附到纸上,完成图像的转印。

⑤ 定影:分离后的纸经定影热辊,碳粉在高温和高压下熔化而永久性地黏附在纸上,实现定影而得到最终的印字输出结果。

⑥ 消除残像:完成转印后,硒鼓表面还留有残余的电荷和碳粉,先经过放电将电荷中和,然后经过清扫辊除去残留的碳粉。这样,硒鼓便恢复原来的状态,以便进行下一次印字过程。

由于激光束扫描速度可以很高,而且打印输出是随硒鼓转动连续进行的,所以打印速度较快,是逐页输出的,因而激光打印机也常称为页式打印设备。

8.10 显示设备

显示设备是将电信号转换成视觉信号的一种装置。在计算机系统中，显示设备被用作输出设备和人机对话的重要工具。与打印机等硬拷贝输出设备不同，显示器输出的内容不能长期保存，当显示器关机或显示别的内容时，原有内容就消失了，所以显示设备属于软拷贝输出设备。

8.10.1 显示器概述

计算机系统中的显示设备，若按显示对象的不同可分为字符显示器、图形显示器和图像显示器。字符显示器是指能显示有限字符形状的显示器。图形和图像是既有区别又有联系的两个概念，图形是指以几何线、面、体所构成的图；而图像是指模拟自然景物的图，如照片等。从显示角度看，它们都是由像素（光点）组成的。如果以点阵方式显示字符，则图形图像显示器也能覆盖字符显示器的功能。事实上目前常用的CRT显示器都具有两种显示方式：字符方式和图形方式，所以它们既是字符显示器，又是图形图像显示器。

若按显示器件的不同显示设备可分为阴极射线管（CRT）、等离子显示器（PD）、发光二极管显示器（LED）、场致发光显示器（ELD）、液晶显示器（LCD）、电致变色显示器（ECD）和电泳显示器（EPID）等。这些显示器件按显示原理可分为两类：一类是主动显示器件，如CRT显示器、发光二极管显示器等，它们是在外加电信号作用下，依靠器件本身产生的光辐射进行显示的，因此也称为光发射器件；另一类称为被动显示器件，如液晶显示器，这类器件本身不发光，工作时需另设光源，在外加电信号的作用下，依靠材料本身的光学特性变化，使照射在它上面的光受到调制，因此这类器件又称为光调制器件。

计算机系统中使用最广泛的是CRT显示器和液晶显示器。CRT显示器具有成本较低、亮度高、色彩鲜明真实、分辨率高、性能稳定可靠等优点；但也存在着体积大、笨重、功耗大等缺点。液晶显示器则体积小、重量轻、功耗低、辐射小，但亮度较低，色彩不够鲜明，且成本较高。随着生产工艺的不断改进，液晶显示器的价格不断下降，逐渐普及，而为普通用户所使用的CRT显示器却已在2010年停产。目前CRT显示器的用途，仅限于一些有特殊需求的专业领域，像军用、医用、航天等方面。但是，CRT显示器具有的可视角度大、无坏点、色彩还原度高、色度均匀、可调节的多分辨率模式、响应时间极短等绝对的优势仍是液晶显示器所无法替代的。

8.10.2 CRT显示器

随着计算机技术的发展和应用的拓展，CRT显示器的发展也很快，从20世纪80年代初到现在，CRT显示器的分辨率已从320×200发展到1024×768，有的达到1280×1024和1600×1200以上；颜色也由单色发展到16M色；显像管的点距从0.6mm以上发展到0.21mm以下；行扫描频率从15.8kHz发展到120kHz以上；显示屏幕尺寸从12in发展到20in以上；显示屏幕也越来越平面化。目前的CRT显示器已朝着高分辨率、高亮度、平面化、大屏幕、低辐射等方向发展。

CRT显示器由显示适配器（显卡）和显示器（监视器）两部分组成，显卡通常插在微机的

总线插槽上，也有的微机主板上集成了显卡电路。显卡到显示器通过显示专用接口连接。图8-15给出了CRT显示器的组成示意图。

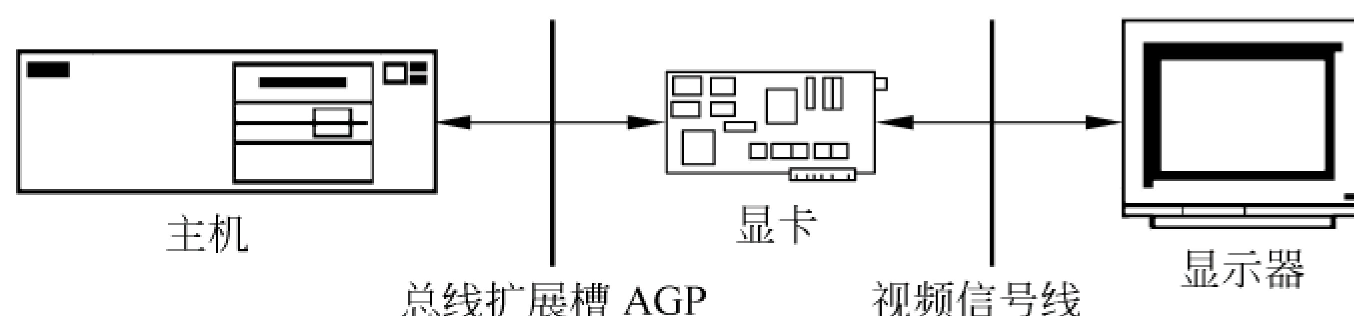


图8-15　CRT显示器的组成

1. CRT显示器的主要技术指标

(1) 点距

点距(Dot Pitch)是指屏幕上两个相邻的同色荧光点之间的距离。点距有实际点距、垂直点距和水平点距的差别，如图8-16所示，严格意义上的点距是指实际点距。点距越小，显示的画面就越清晰、自然和细腻。用显示区域的宽和高分别除以水平点距和垂直点距，即得到显示器在水平和垂直方向上最高可以显示的点数(即极限分辨率)。如果超过这个模式，屏幕上的相邻像素会互相干扰，反而使画面模糊不清。早期的14in显示器的点距分为0.28mm、0.31mm、0.39mm几种规格，目前高清晰度大屏幕显示器通常采用0.20～0.28mm的点距。

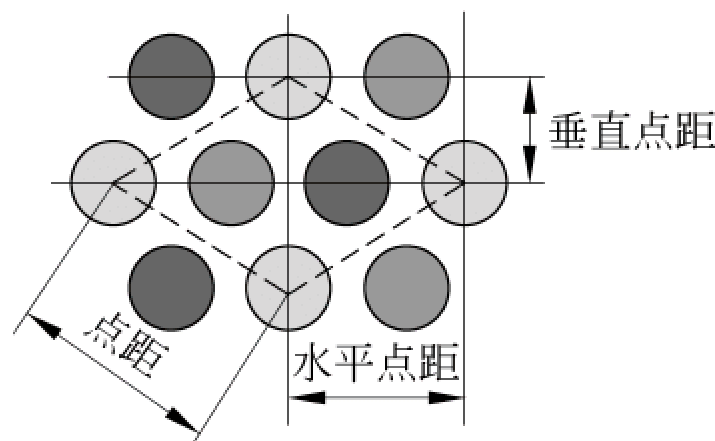


图8-16　CRT显示器的点距

(2) 行频和场频

行频又称水平扫描频率，是电子枪每秒在屏幕上扫描过的水平线条数，以kHz为单位。场频又称垂直扫描频率，是每秒屏幕重复绘制显示画面的次数，以Hz为单位。通常，行频=水平线行数×场频。

由于显示器需要与显卡匹配，所以现在所有的显示器都是变频的(也称多扫描或多频)。频率的范围越大则显示器越贵，其用途也越广。场频决定了图像的稳定性，频率越高越好，典型的场频为50～160Hz，但是它还与分辨率密切相关，如当分辨率为640×480时，某显示器的场频可达到100Hz，而当分辨率为1024×768时，场频将降至60Hz。行频通常为31.5～90kHz或更高，目前比较主流的行频有70kHz、85kHz、96kHz等。

(3) 视频带宽

视频带宽是表示显示器显示能力的一个综合性指标，以MHz为单位。它指每秒扫描的像素个数，即单位时间内每条扫描线上显示的点数的总和。带宽越大表明显示器显示控制能力越强，显示效果越佳。现在主流的CRT显示器的视频带宽都能达到110MHz以上，高档显示器的带宽可达200MHz以上。

$$视频带宽=水平分辨率\times垂直分辨率\times场频\times1.344$$

其中，常数1.344表示电子枪在扫描时扫过水平方向上的像素点数与垂直方向上的像素点数均应当高于理论值，这样才能避免信号在扫描边缘衰减，使图像四周同样清晰。

(4) 最高分辨率

最高分辨率是定义显示器画面解析度的标准,由每帧画面的像素数决定,以水平显示的像素个数×水平扫描线数表示。例如,800×600表示一幅画面水平方向和垂直方向的像素点数分别是800和600。最高分辨率不仅与显示尺寸有关,还受到点距和视频带宽等因素的制约。值得一提的是,一台显示器在75Hz以上的场频下所能达到的分辨率才是它真正的最高分辨率。

(5) 刷新率

刷新率实际上就等于场频,刷新率越高,意味着屏幕的闪烁越小,对人眼睛产生的刺激越小。行频、场频、最高分辨率这几个参数息息相关。一般来说,行频、场频的范围越宽,能达到的最高分辨率也越高,相同分辨率下能达到的最高刷新率也越高。早期显示器只支持50～60Hz的刷新率,现在VESA(视频电子标准协会)规定85Hz为无闪烁的刷新率,从保护眼睛的角度出发,刷新率越高越好。

(6) 屏幕尺寸

指屏幕对角线长度,一般有14in、15in、17in、19in、20in、21in等。

2. CRT显示原理

(1) CRT显示器的扫描方式

CRT显示器如同电视接收机一样,普遍采用光栅扫描方式。在光栅扫描方式中,电子束在水平和垂直同步信号的控制下有规律地扫描整个屏幕。扫描的方法如下:电子束从显示屏的左上角开始,沿水平方向从左向右扫描,到达屏幕右端后迅速水平回扫到左端下一行位置,又从左到右匀速地扫描。这样一行一行地扫描,直到屏幕的右下角,然后又垂直回扫,返回屏幕左上角,重复前面的扫描过程。在水平和垂直回扫时,电子束是"消隐"的,荧光屏上没有亮光显示。这样,在CRT的屏幕上形成了一条条水平扫描线,称为光栅。图8-17为光栅扫描示意图,图中的虚线表示消隐的水平和垂直回扫线。

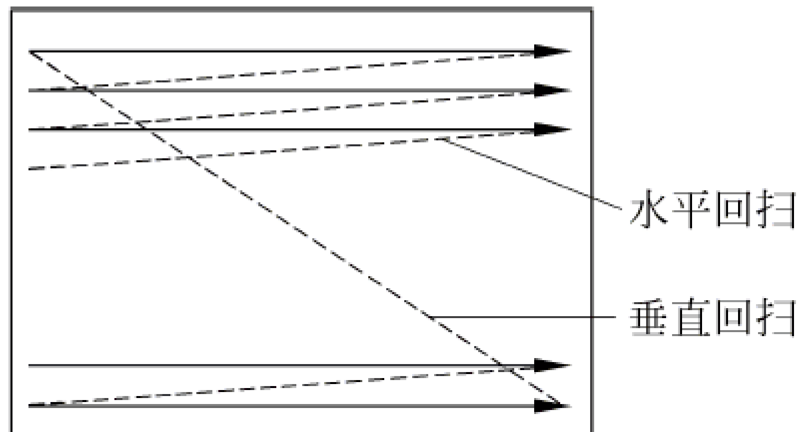


图8-17 光栅扫描

一幅光栅通常也叫做一帧,一帧画面的扫描行数越多,显示出来的画面就越清晰。但要使扫描行数增多,则须使行扫描频率增高,当要求太高时就难于实现。显示器中有两种可能的方法:逐行扫描与隔行扫描。

从上向下依次顺序扫描出所有的行扫描线称为逐行扫描,扫完一场即为一帧。这种方式的控制比较简单,画面质量较好且稳定,但对行扫描频率要求较高。

将一帧画面分为奇数场和偶数场,奇数行组成奇数场,偶数行组成偶数场。第一场显示奇数行,第二场显示偶数行的过程称为隔行扫描,扫描一帧画面需要两场。如果每一帧总行数不变以维持所要求的分辨率,则每一场的行数将减少一半,相应的行扫描频率也将降低一半。由于一帧由两场合成,所以画面质量较逐行扫描方式稍差一些。目前,微机中的显示器一般采用逐行扫描方式。

(2) 显示器的显示模式

显示模式从功能上分为两大类:字符模式和图形模式。

字符模式也称字母数字模式,即A/N模式(Alpha Number Mode)。在这种模式下,显

示缓冲区中存放着显示字符的代码(ASCII码)和属性。显示屏幕被划分为若干个字符显示行和列,如80列×25行。

由于字符模式在CRT显示器上不是点控制,而是一个由8位代码(ASCII码)控制的一块,如像8×8、8×14等大小的显示区域。因此显示缓冲区较小,显示更新的速度非常快,但缺点是无法显示图形。目前流行的所有显卡都包含字符模式。

图形模式也称APA模式(All Points Addressable Mode),即对所有点均可寻址。通常把它称为位图化的显示器,因为屏幕上的每个像素都对应显示缓冲区中的一位或多位。

(3) 显示缓冲区

荧光屏上涂的是中短余辉荧光材料,否则会导致图像变化时前面图像的残影滞留在屏幕上,但如此一来,就要求电子枪不断地反复"点亮""熄灭"荧光点,即便屏幕上显示的是静止图像,也照常需要不断地刷新。

为了不断提供刷新画面的信号,必须把字符或图形信息存储在一个显示缓冲区中,这个缓冲区又称为视频存储器(VRAM)。显示器一方面对屏幕进行光栅扫描,另一方面同步地从VRAM中读取显示内容,送往显示器件。因此,对VRAM的操作是显示器工作的软、硬件界面所在。

VRAM的容量由分辨率和灰度级决定,分辨率越高,灰度级越高,VRAM的容量就越大。同时,VRAM的存取周期必须满足刷新率的要求。

分辨率由每帧画面的像素数决定,而像素具有明暗和色彩属性。黑白图像的明暗程度称为灰度,明暗变化的数量称为灰度级,所以在单色显示器中,仅有灰度级指标。彩色图像是由多种颜色构成的,不同的深浅也可算作不同的颜色,所以在彩色显示器中能显示的颜色种类称为颜色数。如果颜色数较少,不足以逼真地显示图像,则称为伪彩色显示。如果颜色数量多,显示逼真,则称为真彩色显示。真彩色一般要求调色板能达到显示$2^{24}=16M$(1677万)种颜色的能力。

在字符显示方式中,将一屏中可显示的最多字符数称为分辨率。例如,80列×25行表示每屏最多可显示25行,每行可有80个字符。字符方式的VRAM通常分成两部分:字符代码缓存和显示属性缓存。字符代码缓存中存放着显示字符的ASCII码,每个字符占1个字节;显示属性缓存中存放着字符的显示属性,一般也占1个字节。VRAM的最小容量是由屏幕上字符显示的行、列规格来决定的。例如,一帧字符的显示规格为80×25,那么VRAM中的字符代码缓存的最小容量就是2KB。缓存的容量也可以大于一帧字符数,用来同时存放几帧字符的代码。在这种情况下,通过控制缓存的指针就可以在屏幕上显示不同帧中的字符内容,实现屏幕的硬件滚动。

在图形显示方式中,将一屏中可显示的像素点数称为分辨率,图形方式的显示信息以二进制的形式存储在VRAM中,这些信息是图形元素的矩阵数组,在最简单的情况下,只需要存储两值图形,即用"0"表示黑色(暗点),用"1"表示白色(亮点)。用VRAM的1位表示1个点,所以VRAM的1个字节可以存放8个点。例如,一个CRT显示器的分辨率为640×200,在无灰度级的单色显示器中,只需要16KB的VRAM。在彩色显示或单色多灰度显示时,每个点需要若干位来表示。例如,若用两位二进制代码表示1个点,那么每个点便能选择显示4种颜色,但是此时VRAM的1个字节只能存放4个点,如果显示器的分辨率不变,VRAM的容量就要增加一倍。反之,若VRAM容量一定,则随着分辨率的增高,显示

的颜色数将减少。

在图形方式下，VRAM 中用于存放每个像素点颜色信息的位数称为颜色深度或色彩深度，颜色深度与颜色数的对应关系为：

$$颜色深度=\log_2 颜色数$$

8.10.3 字符显示器的工作原理

1. 字符显示原理

字符显示器显示字符的方法也是以点阵为基础的。通常将显示屏幕划分成许多方块，每个方块称为一个字符窗口，它包括字符显示点阵和字符间隔。一般的字符显示器可显示 80 列×25 行=2000 个字符，字符窗口数目为 80×25，如图 8-18 所示。在单色字符显示方式下，每个字符窗口为 9×14 点阵，对应的分辨率为 80 列×25 行(720×350 点阵)，其中字符本身点阵为 7×9，同一字符行中字符横向间隔两个点，不同字符行间的间隔为 5 个点。

屏幕上每个字符窗口对应于 VRAM 中的一个字节单元，在实际的 VRAM 中，还需存入字符的显示属性，所以 VRAM 的容量还需增加一倍。VRAM 中存放的是字符的 ASCII 码，不是点阵信息。若要显示出字符的形状，还要有字符发生器(字符库)的支持。

显示器的字符库是用来存放各种字符的点阵字形辉亮数据的只读存储器。显示时，从 ROM 中读出有关的点阵信息送给 CRT 作为辉亮控制信号，以控制电子束的强弱，从而在屏幕上组成字符。显示器的字符库中存放的是字符的行点阵码，字符库的高位地址来自 VRAM 的 ASCII 码，低位地址来自行计数器的输出 $RA_3 \sim RA_0$(行扫描线序号)。图 8-19 给出了字符“A”的点阵字形，这是一个 7×9 的点阵，用二进制码中的“1”对应屏幕上的亮点，“0”对应暗点。对于字符“A”可用 9 个字节的行点阵码表示，从第一行到第九行分别为 10H、28H、44H、82H、82H、FEH、82H、82H、00H。从字符库中读出行点阵码，就能显示出该字符。

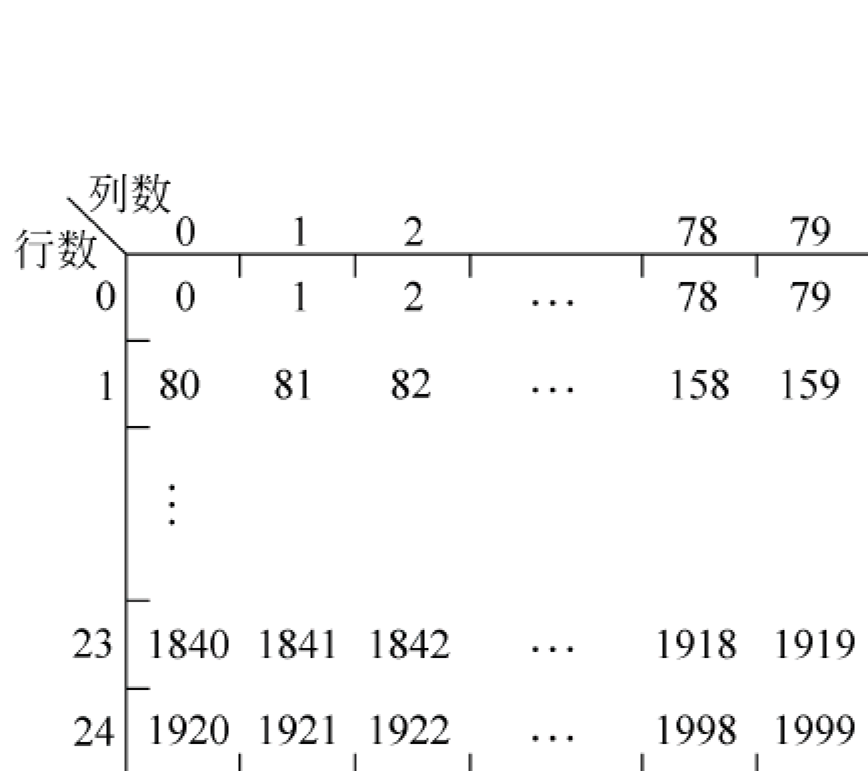


图 8-18　屏幕上字符位置的分配

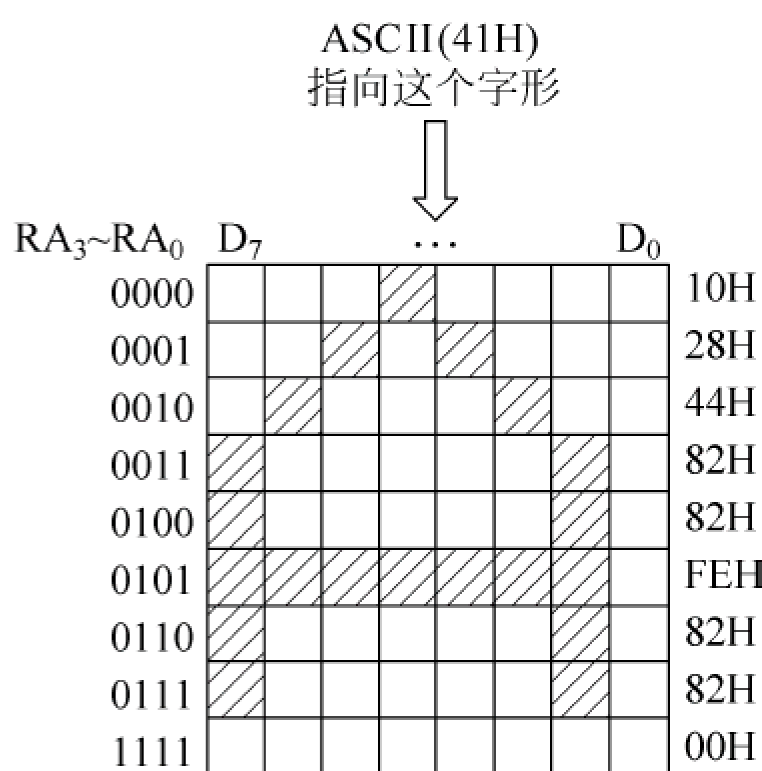


图 8-19　“A”的点阵字形

在屏幕上，每个字符行要显示多个字符，而电子束在光栅扫描时，采用的是逐行扫描法。按照这种扫描法，在显示字符时，并不是对显示的每个字符单独进行点阵扫描(即扫描完一个字符的各行点阵，再扫描另一个字符的各行点阵)，而是采用对一排所有字符的点阵进行逐行依次扫描。例如，某字符行欲显示的字符是 A，B，C，…，T，显示电路首先根据各字符代

码依次从字符发生器取出A,B,C,…,T各个字符的第一行点阵代码,并在字符行第一条扫描线位置上显示出这些字符的第一行点阵;然后再依次取出该排各个字符的第二行代码,并在屏幕上扫出它们的第二行点阵。如此循环,直到扫描完该字符行的全部扫描线,那么每个字符的所有点阵(如9行点阵)便全部显示在相应的位置上,屏幕上就出现了一排完整的字符。当显示下一排字符时,重复上述的扫描过程。

2. VRAM的地址组织

在字符显示器中,屏幕上每个字符位置对应VRAM中的一个字节,VRAM中各字节单元的地址随着屏幕由左向右,自上而下的显示顺序从低向高安排。也就是说,VRAM的0号单元存放的字符代码经字符发生器转换为字形点阵后,显示在屏幕第一行字符左边第一个位置上;1号单元存放的字符代码转换后显示在屏幕第一行左边的第二个位置上……VRAM的最后一个单元存放的字符代码转换后显示在屏幕最后一行右边最末一个位置上。VRAM的地址安排与屏幕位置的对应关系如图8-20所示。

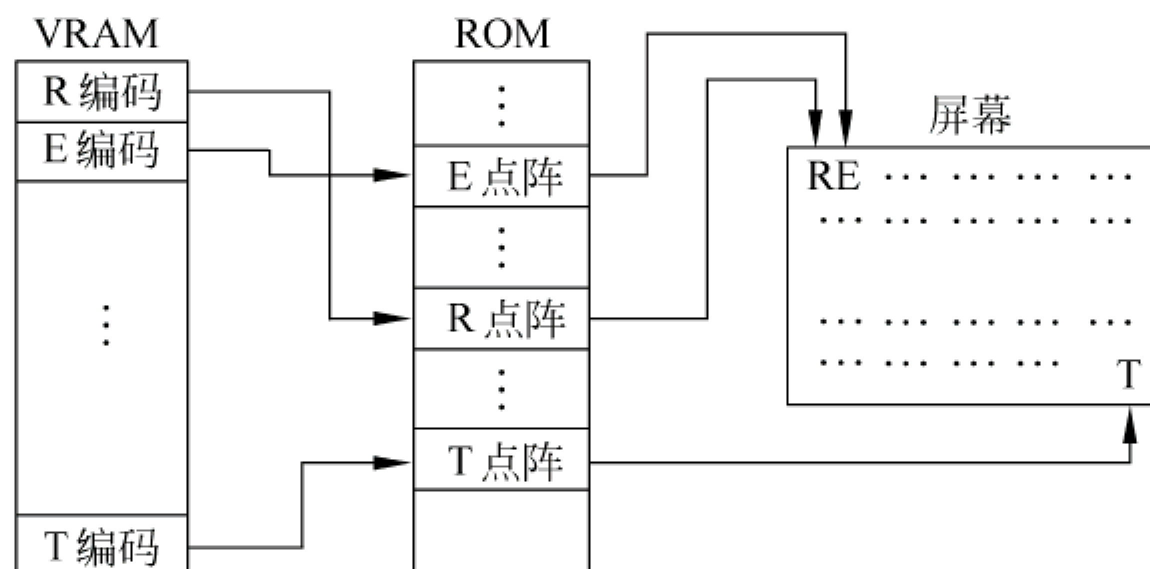


图8-20　VRAM的地址安排与屏幕位置的对应关系

3. 字符显示器的控制电路

图8-21是字符显示器的定时控制电路。它的核心是点计数器、字计数器(水平地址计数器)、行计数器和排计数器(垂直地址计数器),由它们来控制显示器的逐点、逐字、逐行、逐屏幕的刷新显示。为了避免扫描行和字符行这两个概念的混淆,在下例中把扫描行仍称为行,而把字符行称为排。

每次从字库中读出一行字符点数据7位,送入移位寄存器,然后在点脉冲控制下串行地移位输出,送往显示器作为亮度控制信号:“1”亮,“0”暗。移位寄存器实现并—串转换,每发一个点脉冲,屏幕上产生一个像点。

点计数器对一个字符的列数和字符横向间隔进行计数,为9分频,即输入9个点脉冲后完成一次计数循环,并向下一级计数器输出一个计数脉冲,这对应于一个字符横向7点,加上两点间距。

字计数器用来同步控制一条水平扫描线的正扫和回扫。由于一排可有80个字符,需在扫描正程中显示,所以当字计数器由0计到79时,光栅从左向右扫满一行。然后进入回扫逆程,设逆程需占18个字符扫描时间(折合值),因此字符计数器为(80+18)=98分频,即每输入98个计数脉冲完成一个计数循环。

行计数器对字符窗口的高度进行控制,字符窗口的高度所占的扫描线数为14。CRT

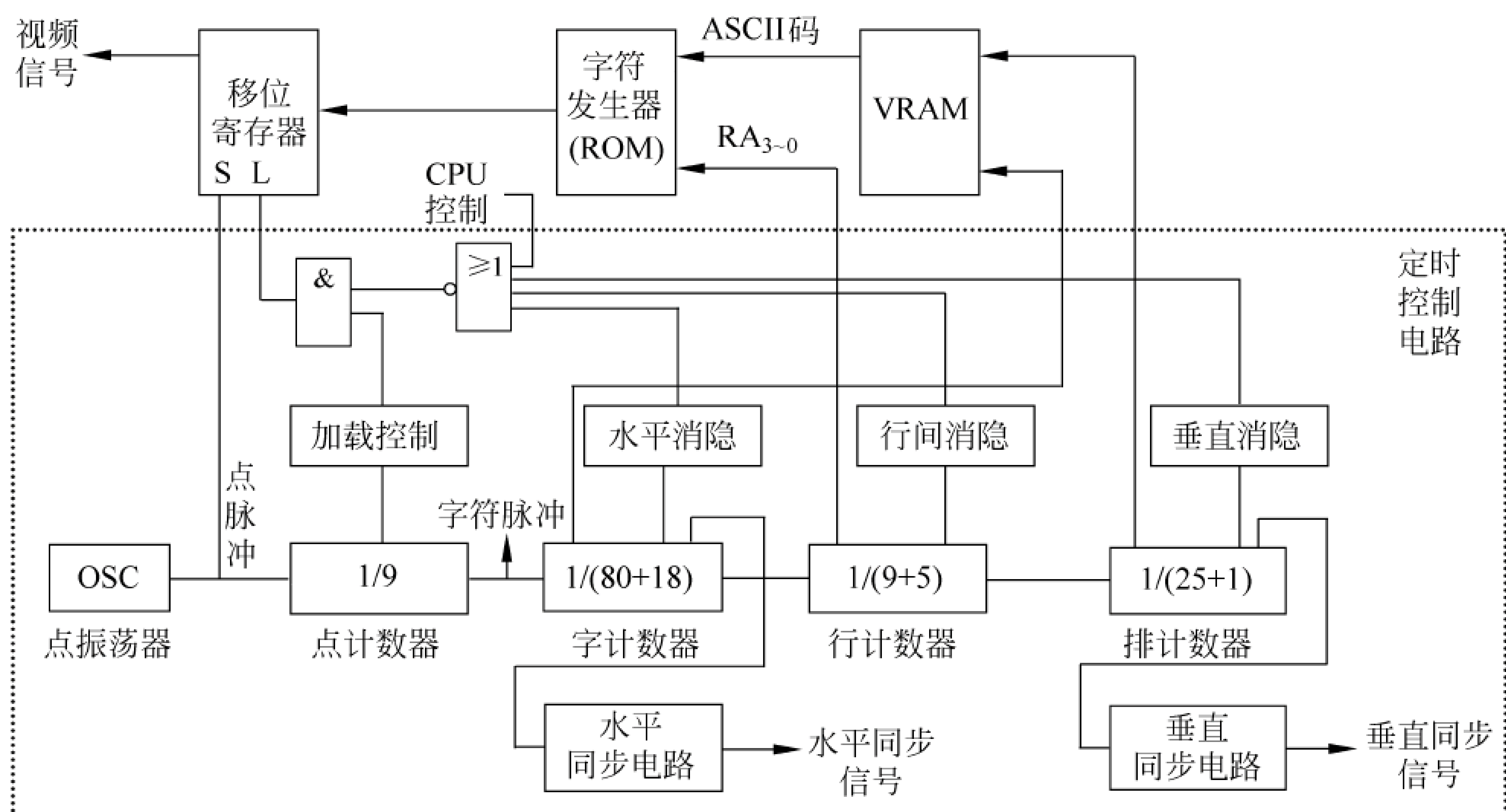


图 8-21　字符显示器定时控制电路

每完成一次水平扫描，只能显示一排字符中的一行。只有依次扫描9行后，才能完整地显示出一排字符，再扫描5行并消隐之后，即形成排间的空白间距。所以行计数器为(9+5)=14分频。

排计数器对应于屏幕的垂直扫描及其回扫。正程显示25排字符，当排计数器从0计数到24时，光栅正好从上向下扫完一屏，然后进入回扫逆程，回到屏幕左上角。逆程时间等于扫描一排字符的时间，折合值为1，所以计数分频值为(25+1)=26。

显然排计数值体现了当前显示字符的排号，字计数值体现了当前显示字符的列号，它们决定了字符的显示位置。因此由排、列号可转换为VRAM的地址，据此找到对应的单元，取出字符代码(ASCII码)。该字符代码作为字符库的高位地址，而行计数值作为低位地址，据此可读出该字符点阵的对应行数据，经移位寄存器串行输出，放大后驱动CRT控制栅极，决定像点的亮度。

字计数器的一个循环，启动CRT行扫描电路开始新的一行水平扫描。排计数器的一个循环，启动CRT场扫描电路开始新的一场扫描。

8.10.4　图形显示器的工作原理

下面以某彩色图形显示器为例，介绍图形显示的基本原理。设该彩色图形显示器的分辨率为640×480，可同时显示16种颜色。VRAM中存放着显示的图形点阵数据，由于计算机只能以二进制方式存放数据，每位只有两种状态(“0”或“1”)。对于单色显示，VRAM中的每一位对应画面上的一个像素点，该位为“1”即表示画面上的这一点是亮点。而对于彩色显示(如16种颜色)，就需要用VRAM中的4位来定义一种颜色。在彩色图形显示器中经常采用彩色位平面的存储结构来表示颜色信息。每个彩色位平面由单一位组成，并表示屏上某个可以显示的颜色。例如，分辨率为640×480，每个位平面含有640×480位，即有307 200位的信息。由于要同时显示16种不同颜色，它就具有4个彩色位平面，故需要1 228 800位的VRAM，即153 600B。所以，VRAM的总容量=640×480×4b≈150KB。它

被分为4个位平面,每个位平面提供彩色代码中的一位,每个位平面的容量为37.5KB。

从屏幕显示角度,每一行由4个位面中的80个字节来表示(640/8=80)。屏幕上的一个彩色像素点,需要用来自4个位平面上每个位平面的相同位置的一个存储位表示。

根据上述对应关系,可设计出显示器控制逻辑中的同步计数分频关系,如图8-22所示。

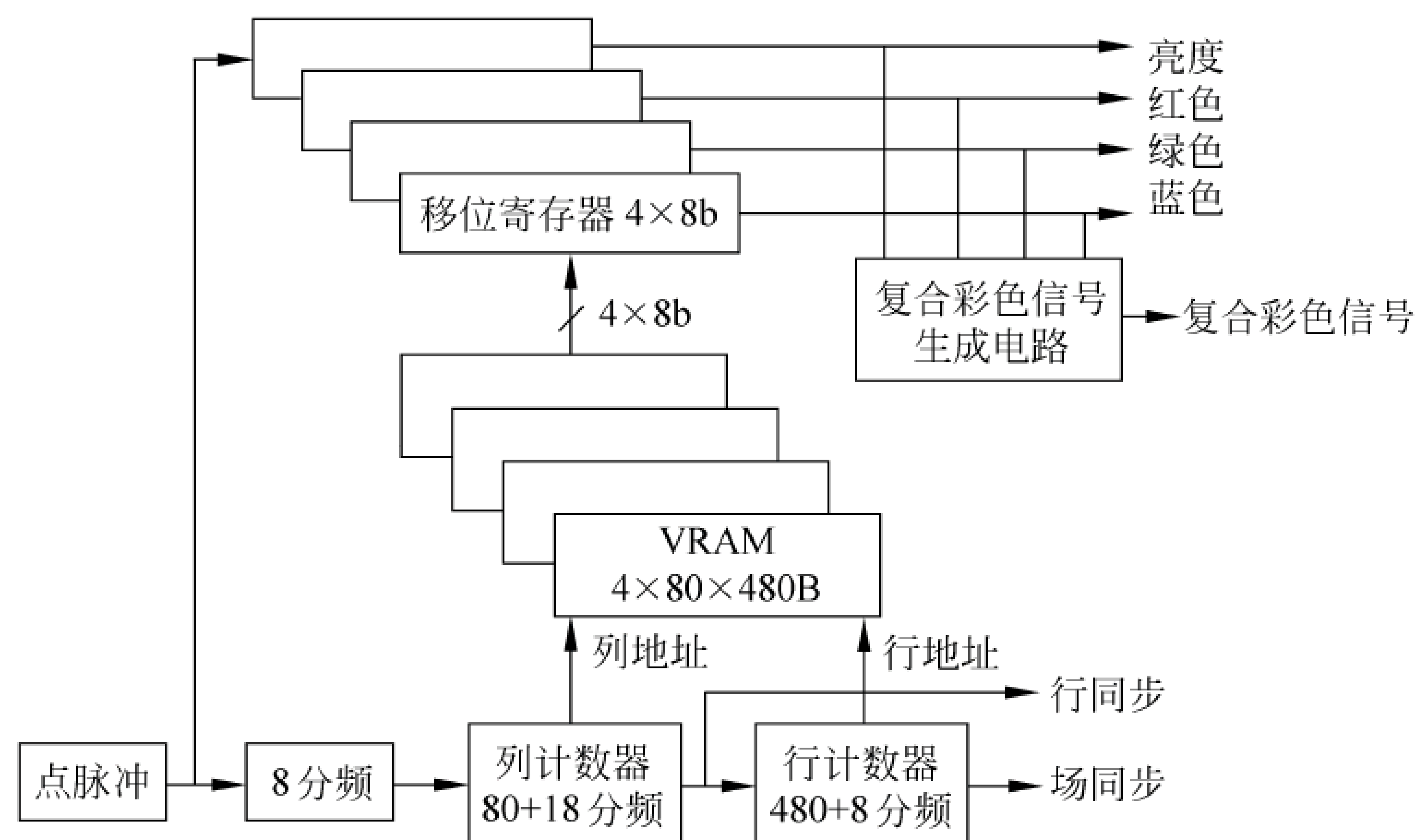


图8-22 彩色CRT控制逻辑原理

图形/图像以像素为单位,但在VRAM中以字节为单位按地址存储,即将一条水平线上自左向右,每8个点的代码作为一个字节,存放在一个编址单元中。因此,点脉冲经点计数器8分频之后产生字节脉冲,每发一次字节脉冲就访问一次VRAM,从4个位平面中各读出一个字节(8点)送往移位寄存器,再串行输出形成亮度信号与红、绿、蓝三色信号,它们的组合决定了16色中的一种。若用于单色显示器,则将4位代码转换为16级亮度调制信号,用于控制像素的灰度。

列计数器又称字节计数器,98分频。计数值从0到79,光栅从左向右扫描一行,正程显示80个字节共640点。字节计数器所附加的18次计数作为行线逆程回扫时间,逆程回扫应当消隐。

行计数器为488分频。计数值从0到479,对应于场正程扫描,显示480行;附加8次计数,对应于场逆程回扫,逆程回扫应消隐。

行计数值与列计数值决定了屏幕当前显示位置(8点一组),相应的VRAM地址为:行号×80+列号。按该地址同时访问4个位平面,取出4个字节的图形代码。列计数一个循环,输出一个行扫描同步信号;行计数一个循环,输出一个场扫描同步信号。这就使得对VRAM的访问与CRT的扫描严格同步,能获得稳定的显示画面。

从以上的分析可以看出,分辨率、颜色数与VRAM容量密切相关。对于字符显示方式,如果分辨率为c列×l行,而一个字符的编码与属性、颜色数共需占n字节,则VRAM的总容量应不少于$c \times l \times n$字节。对于图形显示方式,如果分辨率为$c \times l$像素,而每个像素的颜色数用n位二进制代码表示,则VRAM容量应不少于$c \times l \times n$位。两种显示方式的c、l值不同,显然,图形方式所需的VRAM容量一般都大于字符方式。如果一台CRT显示器既可用作字符方式又可用作图形方式,且各有数种分辨率规格,则VRAM的容量计算应以最高分辨率图形方式为准。

一台显示器可以显示的字符种类与字符点阵规格，决定了字符发生器ROM的容量大小，而VRAM的容量与此无关。

8.10.5 LCD显示器

LCD(Liquid Crystal Display)就是液晶显示器，LCD有低眩目的全平面屏幕，需要的功率很低，有源阵列的LCD面板的色彩质量实际上超过了大多数CRT显示器。

1. LCD显示原理

LCD显示器提供比同尺寸CRT显示器更大的可视图像，有4种基本的LCD选择：无源阵列单色、无源阵列彩色、有源阵列模拟彩色和最新的有源阵列数字彩色。无源阵列的单色和彩色显示屏主要是用作早期低档便携式计算机的显示器或者工业用的桌面显示面板，与有源阵列模块相比，具有相对较低的价格和较强的耐用性。

大多数通用无源阵列显示器采用超级偏转向列型设计，因此这些面板经常被称为STN(Super Twist Nematic)。有源阵列显示器采用薄膜晶体管设计，因此被称为TFT(Thin Film Transistor)。

在LCD中有两个偏振器，偏振器只允许与其方向相同的光波通过，经过偏振器后的光波都成同一方向。通过改变第二个偏振器的角度，允许通过的光数量可以改变。改变偏振角和控制通过的光数量，就是液晶单元所扮演的角色。在彩色LCD中，另有一个附加偏振器为每个像素分配3个单元，分别显示红、绿、蓝中的一种。

液晶单元是像液体一样可以流动的棒状分子，可以使光线直接通过，但是电荷可以改变晶体的方向及通过它的光线的方向。尽管单色LCD没有彩色偏振器，但是每个像素有多个单元来控制灰度的深浅。

在一个无源阵列的LCD中，每个液晶单元被两个晶体管的电荷所控制，它取决于晶体在屏幕上的行列位置。沿着屏幕水平和垂直边缘的晶体管数目决定了屏幕的分辨率。例如，一个具有1024×768分辨率的屏幕，在水平边界有1024个晶体管，在垂直边缘上有768个晶体管，总共有1792个。当液晶单元响应自己的两个晶体管的脉冲时，将对光波产生偏转，电荷越强，光波偏转得越厉害。

在无源阵列LCD中的电荷是脉冲式的，所以显示器缺少像有源阵列那样的亮度，为了增加亮度，现采用一种称为双扫描的新技术，将无源阵列屏幕分为上半部和下半部，让两个独立电路同时驱动显示器的上半部和下半部，减少每个脉冲之间的间隔时间。除了增加亮度，双扫描设计还提高了响应速度，使这种类型对于全动态视频或其他显示信息快速变化的应用更有用处。

在有源阵列LCD中，每个单元在显示屏之后有自己专用的晶体管，对其充电进而偏转光波。于是，一个1024×768的有源阵列显示器就有786 432个晶体管。提供比无源阵列显示器更亮的图像，因为各单元能够维持一个恒定的、较长时间的充电。然而，有源阵列技术的能耗比无源阵列的大，有源阵列显示器制造起来比较困难，价格更高。

在有源和无源阵列LCD中，第二个偏振器控制通过每个单元的光量。这些单元把光线的波长偏转到接近匹配偏振器允许的波长。每个单元通过偏振器的光量越多，像素越亮。

单色LCD显示器通过改变单元的亮度或者以开关模式高频振动单元来获得灰度级别

(可到64级),而彩色LCD高频振动3个彩色单元,并控制它们的亮度以获得屏幕上的不同颜色。

超偏转和三重超偏转LCD技术的出现使得用户能够从更大的角度,以更好的对比度和亮度清晰地观看屏幕。为了在微光的情况下改善清晰度,一些便携机加入了背光和侧光(也称为边光)。背光屏幕从LCD后面的面板获取光线,侧光屏幕从安装在屏幕边缘上的小的荧光管获取光线。

目前,最好的彩色显示器是有源阵列TFT LCD,其中每个像素都由3个晶体管驱动和控制(红、绿、蓝),因此可以精确地控制每一个像素,获得高质量的图像。

2. LCD的技术指标

由于显示原理与传统CRT显示器的根本不同,因此CRT显示器的耗电大、体积大、有辐射、有闪烁等弊端在LCD上将不复存在,LCD的技术指标也有一些变化。

(1) 像素间距

LCD的像素间距类似于CRT显示器的点距,但LCD的像素间距对于产品性能的重要性远没有CRT的点距那么高。因为LCD的像素数量是固定的,在尺寸与分辨率都相同的情况下,大多数LCD的像素间距基本相同,主流的LCD像素间距在0.3mm左右。

(2) 分辨率

由于LCD的像素间距固定,所以分辨率不能任意调整。LCD只有在最佳分辨率下,才能显现出最佳影像。要确定一款显示器的最佳分辨率,单单根据显示器的尺寸是无法确定的,需要综合考虑屏幕比例、屏幕尺寸及物理像素才能确定出最佳分辨率,而且显卡的性能也决定着可设置分辨率的设置范围。目前,常见的显示器屏幕比例(长∶宽)有4种:5∶4、4∶3、16∶10和16∶9。例如,4∶3对应的分辨率有1024×768,5∶4对应的分辨率有1280×1024,16∶9对应的分辨率有1920×1080等。更大尺寸拥有更大的最佳分辨率。在呈现其他的分辨率显示模式时只能以扩展或压缩的方式将画面显示出来,显示效果将受到影响。

(3) 可视角度

可视角度是指人们清晰观察显示屏幕的范围,这是LCD的一个重要的指标,因为LCD从侧面观看时,亮度、对比度都会有明显的下降。可视角度参数可用水平(左右)、垂直(上下)来衡量,也可以用左/右、上/下分别来衡量。

(4) 亮度

由于LCD是被动式发光,因此在亮度、对比度方面的指标可能不如主动发光的CRT显示器。LCD的亮度取决于LCD的结构和背景照明的类型。亮度的测量单位通常为坎德拉每平方米(cd/m^2),LCD的亮度普遍在200~500cd/m^2。

(5) 对比度

对比度实际上就是亮度的比值,即白色画面(最亮时)下的亮度除以黑色画面(最暗时)下的亮度。在合理的亮度值下,对比度越高,其所能显示的色彩层次越丰富。传统的对比度又称为静态对比度,现在主流的显示器静态对比度一般为1000∶1~1500∶1。动态对比度就是在原有基础上加进了一个自动调整显示亮度的功能,这样就将原有对比度提高了几倍甚至几十倍,但本质上真正的对比度没有改变,所以画面细节并不会显示得更清晰,但因为它的自动调节亮度的功能而在很多游戏中会有比较好的表现。

(6) 响应时间

响应时间反映了液晶显示器各像素点对输入信号反应的速度，即每个像素由暗转亮或由亮转暗所需要的时间。响应时间一般被分为上升时间和下降时间，而表示时应以两者之和为准。从早期的25ms到目前的几毫秒，甚至1ms，响应时间在不断地缩短，响应时间越短则使用者在看动态画面时越不会有尾影拖曳的感觉。

(7) 色彩数

色彩数就是屏幕上最多显示多少种颜色的总数。目前，LCD的液晶板有8位和6位两种，前者由红绿蓝三原色每种颜色8位色彩组成，组合起来就是24位真彩色，这种LCD的颜色一般标称为16.7M；后者三原色每种只有6位色彩，液晶板通过“抖动”技术，局部快速切换相近颜色，利用人眼的残留效应获得缺失色彩，这种LCD的颜色一般标称为16.2M。这是因为抖动技术不能获得完整的256色效果，通常只有253色，3个253相乘就是16.2M色。不过两者实际视觉效果差别不算太大，目前高端LCD以16.7M色占主流。

8.10.6 视频显示标准

PC系列微机的显示系统由显示器和显示适配器(显卡)构成，显示器和显卡必须配套使用。下面介绍PC系列微机的几种显示标准。

1. MDA

MDA(Monochrome Display Adapter)属于单色显示适配器，是IBM最早研制的视频显示适配器。MDA支持80列、25行字符显示，采用9×14点阵的字符窗口，对应的分辨率为720×350。MDA的字符显示质量高，但是不支持图形功能，也无彩色显示能力。

2. CGA

在MDA推出的同时，IBM也推出了彩色图形适配器(Color Graphics Adapter，CGA)。CGA支持字符、图形两种方式，在字符方式下又有80列、25行和40列、25行两种分辨率，但字符窗口只有8×8点阵，故字符质量较差。在图形方式下，有640×200和320×200两种分辨率，在最高分辨率的图形显示方式下的颜色数可达4种。

3. EGA

增强的图形适配器(Enhanced Graphics Adapter，EGA)是IBM公司推出的第二代图形显示适配器，它兼容了MDA和CGA全部功能。EGA的显示分辨率达到640×350，字符显示窗口为8×14点阵，使字符显示质量大大优于CGA而接近于MDA。在最高分辨率的图形显示方式下的颜色数可达16种。

4. VGA

视频图形阵列(Video Graphics Array，VGA)是IBM公司推出的第三代图形显示适配器，它兼容了MDA、CGA和EGA的全部功能。VGA的显示分辨率为640×480，可显示256种颜色。近年来又出现了超级VGA(SVGA)。在VGA中，显示颜色由D/A转换的输出位数和调色板的位数决定。其标准是：红、绿、蓝每一路视频信号均采用6位D/A转换，

并使用18位的彩色调色板,因此最多可以组合出$2^{18}=256K$种颜色。但是,每次可以同时显示的颜色数还取决于每个像素在VRAM中的位数。在分辨率为640×480时,每个像素对应4位信息,因此可以从256K种颜色中选择16种颜色;在分辨率为320×200时,每个像素对应8位信息,可以从256K种颜色中选择256种颜色。VGA的字符显示功能也比EGA有所改进,字符窗口为9×16点阵。

5. TVGA

TVGA是美国Trident Microsys Tems公司开发的超级VGA标准,与VGA完全兼容。分辨率有640×480、800×600、1024×768、1280×1024等,可显示的颜色数有16色、256色、64K色和16M色等。

6. XGA

XGA(eXtended Graphics Array)是IBM公司继VGA之后推出的扩展图形阵列显示标准。其中的配置有协处理器,属于智能型适配器。XGA可实现VGA的全部功能,但运行速度比VGA快。

8.10.7 微型计算机的显示适配器

1. 独立显卡和集成显卡

显示适配器俗称显卡,是微机中进行数模信号转换的设备,承担输出显示图形的任务。显卡接在主板上,它将微机的数字信号转换成模拟信号让显示器显示出来,同时显卡还有图像处理能力,可协助CPU工作,提高整体的运行速度。目前,台式微型计算机有两类显卡可供选择:独立显卡和集成显卡。

独立显卡上有自己的显示核心芯片(GPU)和显存,不占用CPU和主存,其优点是处理数据速度快,缺点是功耗比较高,且需要额外投资购买显卡。

集成显卡是指芯片组内集成了显示核心芯片,使用这种芯片组的主板可以在不需要独立显卡的情况下实现普通的显示功能。集成的显卡不带显存,使用系统的一部分主存作为显存,具体的容量可以由系统根据需要自动动态调整。显然,如果使用集成显卡运行需要大量占用显存的程序,对整个系统的影响会比较明显,此外,由于系统主存的频率通常比独立显卡上显存的频率低很多,因此集成显卡的性能比独立显卡要差。

2. 显卡性能三要素

在决定显卡性能的三要素中,首先是其所采用的显示芯片,其次是显存带宽(这取决于显存位宽和显存频率),最后才是显存容量。显示芯片指的是提供显示功能的芯片,即图形处理器(GPU)。显存位宽是显存在一个时钟周期内所能传送数据的位数,位数越大则相同频率下所能传输的数据量越大,目前的显存位宽至少为64b,高端的可达到2048b甚至更高。显存容量的大小决定着显存临时存储数据的能力,早期显存容量的只有512KB,目前市面上的显存容量已达到1GB以上,甚至已有24GB显存的显卡了。显存容量曾经是影响最大分辨率的一个瓶颈,但目前早已经不再是影响最大分辨率的因素。一款显卡究竟应该

配备多大的显存容量才合适是由其所采用的显示芯片所决定的，也就是说显存容量应该与显示核心的性能相匹配才合理，显示芯片性能越高，其所配备的显存容量相应也应该越大，而低性能的显示芯片配备大容量显存对其性能是没有任何帮助的。

现在决定最大分辨率的其实是显卡的RAMDAC(Random Access Memory Digital Analog Convertor)频率，RAMDAC即随机存取内存数字-模拟转换器，它的作用是将显存中的数字信号转换为显示器能够显示出来的模拟信号，其转换速率以MHz表示。该数值决定了在足够的显存下，显卡最高支持的分辨率和刷新率。如果要在1024×768的分辨率下达到85Hz的刷新率，RAMDAC的速率至少是1024×768×85Hz×1.344(折算系数)≈90MHz。目前，主流显卡的RAMDAC能达到了350MHz和400MHz，已足以满足和超过目前大多数显示器所能提供的分辨率和刷新率。

习　题

8-1　外部设备有哪些主要功能？可以分为哪些大类？各类中有哪些典型设备？

8-2　分别用RZ、NRZ、NRZ-1、PE、FM、MFM和M^2FM制记录方式记录下述数据序列，画出写电流波形。

(1) 1101110001100

(2) 1010110011000

8-3　若对磁介质存储器写入数据序列10011，画出不归零-1制、调相制、调频制和改进的调频制等记录方式的写电流波形。

8-4　主存储器与磁介质存储器在工作速度方面的指标有什么不同？为什么磁盘存储器采用两个以上的指标来说明其工作速度？

8-5　某磁盘组有6片磁盘，每片可有两个记录面，存储区域内径为22cm，外径为33cm，道密度40道/厘米，位密度400b/cm，转速2400r/min。问：

(1) 共有多少个存储面可用？

(2) 共有多少个圆柱面？

(3) 整个磁盘组的总存储容量有多少？

(4) 数据传送率是多少？

(5) 如果某文件长度超过一个磁道的容量，应将它记录在同一存储面上还是记录在同一圆柱面上？为什么？

(6) 如果采用定长信息块记录格式，直接寻址的最小单位是什么？寻址命令中如何表示磁盘地址？

8-6　某磁盘存储器的转速为3000r/min，共有4个盘面，5道/mm，每道记录信息12 288B，最小磁道直径为230mm，共有275道。问：

(1) 该磁盘存储器的存储容量是多少？

(2) 最高位密度和最低位密度是多少？

(3) 磁盘的数据传送率是多少？

(4) 平均等待时间是多少？

8-7　假定某磁盘的转速是12 000r/min，平均寻道时间为6ms，传输速率为50MB/s，有关控制器的开销是1ms，请计算出连续地读写256个扇区(每一扇区大小为512B)所需要的平均时间(忽略扇区间可能有的间隔)。

8-8　某磁盘组有效盘面20个，每个盘面上有800个磁道。每个磁道上的有效记忆容量为13 000B，块

间隔235B,旋转速度3000r/min。问:

(1) 在该磁盘存储器中,若以1000B为一个记录,这样,一个磁道能存放10个记录。若要存放12万个记录,需要多少个圆柱面(一个记录不允许跨越多个磁道)?

(2) 该磁盘存储器的平均等待时间是多少?

(3) 数据传送率是多少?

8-9 某磁盘格式化为24个扇区和20条磁道。该盘能按需要选择顺时针或逆时针旋转,旋转一圈的时间为360ms,读一块数据的时间为1ms。该片上有3个文件:文件A从磁道6、扇区1开始占有两块;文件B从磁道2、扇区5开始占有5块;文件C从磁道5、扇区3开始占有3块。

问:该磁盘的平均等待时间为多少?平均寻道时间是多少?若磁头移动和磁盘转动不同时进行,且磁头的初始位置在磁道0、扇区0,按顺序C、B、A读出上述3个文件,总时间是多少?在相同的初始位置情况下,读出上述3个文件的最短时间是多少?此时文件的读出次序应当怎样排列?

8-10 什么是光盘?简述光盘的工作原理。

8-11 键盘属于什么设备?它有哪些类型?如何消除键开关的抖动?简述非编码键盘查询键位置码的过程。

8-12 针式打印和字模式打印有何不同?各有什么优缺点?

8-13 什么是分辨率?什么是灰度级?它们各有什么作用?

8-14 某字符显示器,采用7×9点阵方式,每行可显示60个字符,缓存容量至少为1260B,并采用7位标准编码,问:

(1) 如改用5×7字符点阵,其缓存容量为多少?(设行距、字距不变——行距为5,字距为1)

(2) 如果最多可显示128种字符,上述两种显示方式各需多大容量的字符发生器ROM?

8-15 某CRT显示器可显示64种ASCII字符,每帧可显示64列×25行,每个字符点阵为7×8,即横向7点,字间间隔1点,纵向8点,排间间隔6点,场频50Hz,采用逐行扫描方式。问:

(1) 缓存容量有多大?

(2) 字符发生器(ROM)容量有多大?

(3) 缓存中存放的是字符的ASCII码还是字符的点阵信息?

(4) 缓存地址与屏幕显示位置如何对应?

(5) 设置哪些计数器以控制缓存访问与屏幕扫描之间的同步?它们的分频关系如何?

8-16 某CRT字符显示器,每帧可显示80列×20行,每个字符是7×9点阵,字符窗口9×14,场频为50Hz。问:

(1) 缓存采用什么存储器,其中存放的内容是什么?容量应为多大?

(2) 缓存地址如何安排?若在243号单元存放的内容要显示出来,其屏幕上X和Y的坐标应是多少?

(3) 字符点阵存放在何处?如何读出显示?

(4) 计算出主振频率以及点计数器、字计数器、行计数器、排计数器的分频频率。

8-17 若用CRT作图形显示器,其分辨率为640×200,沿横向每8点的信息存放在缓存中,场频为60Hz。问:

(1) 缓存的基本容量是多少?

(2) 地址如何安排?

(3) 点计数器、字节计数器、行计数器各为多少分频?

(4) 它和字符显示器有哪些不同?

8-18 某字符显示器分辨率为40列×25行,字符点阵5×7,横向间隔2点,排间间隔4点,问:缓存VRAM容量至少应多大?应设置哪几级同步计数器?它们的分频关系如何?若要求场频60Hz,则点频应为多少?何时访问一次VRAM?地址如何确定?

8-19 某图形显示器的分辨率为800×600,若作单色显示且不要求灰度等级,则VRAM容量至少应

多大？应设置哪几级同步计数器？它们的分频关系如何？若要求场频 60Hz，则点频应为多少？何时访问一次 VRAM？地址如何确定？

8-20　某图形显示器的分辨率为 640×480，刷新频率为 50Hz，且假定水平回扫期和垂直回扫期各占水平扫描周期和垂直扫描周期的 20%，试计算图形显示器的行频、水平扫描周期、每个像素的读出时间和视频带宽。若分辨率提高到 1024×768，刷新频率提高到 60Hz，再次计算图形显示器的行频、水平扫描周期、每个像素的读出时间和视频带宽。

8-21　水平扫描频率(行频)的单位为 kHz，垂直扫描频率(场频)的单位为 Hz，两者为何相差 1000 倍？

第9章 输入输出系统

计算机的输入输出系统是整个计算机系统中最具有多样性和复杂性的部分，本章首先介绍主机与外设之间的连接问题，接着重点介绍程序查询方式、程序中断方式、DMA方式和通道方式4种输入输出控制方式。

9.1 主机与外设的连接

现代计算机系统中外部设备的种类繁多，各类外部设备不仅结构和工作原理不同，而且与主机的连接方式也是复杂多变的。

9.1.1 输入输出接口

主机和外设的连接方式有辐射型连接、总线型连接等。输入输出接口(I/O接口)是主机和外设之间的交接界面，通过接口可以实现主机和外设之间的信息交换。

主机和外设之间进行信息交换为什么一定要通过接口呢？这是因为主机和外设各自具有自己的工作特点，它们在信息形式和工作速度上具有很大的差异，接口正是为了解决这些差异而设置的。下面首先分析主机和外设之间需要交换的信息。

1. 数据信息

这类信息可以是通过输入设备送到计算机的输入数据，也可以是经过计算机运算处理和加工后，送到输出设备的结果数据。传送可以是并行的，也可以是串行的。

2. 控制信息

这类信息是CPU对外设的控制或管理命令，例如外设的启动和停止控制、输入或输出操作的指定、工作方式的选择、中断功能的允许和禁止等。

3. 状态信息

这类信息用来标志外设的工作状态，例如输入设备数据准备好标志、输出设备忙、闲标志等。CPU在必要时可通过对它的查询来决定下一步的操作。

4. 联络信息

这类信息是主机和外设间工作的时间配合信息,通过联络信息可以决定不同工作速度的外设和主机之间交换信息的最佳时刻,以保证整个计算机系统能统一协调地工作。

5. 外设识别信息

这类信息是I/O寻址的信息,使CPU能从众多的外设中寻找出与自己进行信息交换的唯一的设备。

9.1.2 接口的功能和基本组成

1. 接口的功能

(1) 实现主机和外设的通信联络控制

接口中的同步控制电路用来解决主机与外设的时间配合问题。

(2) 进行地址译码和设备选择

当CPU送来选择外设的地址码后,接口必须对地址进行译码以产生设备选择信息,使主机能和指定外设交换信息。

(3) 实现数据缓冲

数据缓冲寄存器用于数据的暂存,以避免丢失数据。在传送过程中,先将数据送入数据缓冲寄存器中,然后再送到输出设备或主机中。

(4) 数据格式的变换

为了满足主机或外设的各自要求,接口电路中必须具有各类数据相互转换的功能。例如,并-串转换、串-并转换、模-数转换、数-模转换以及二进制数和ASCII码的相互转换等。

(5) 传递控制命令和状态信息

当CPU要启动某一外设时,通过接口中的命令寄存器向外设发出启动命令;当外设准备就绪时,则有"准备好"状态信息送回接口中的状态寄存器,为CPU提供外设已经具备与主机交换数据条件的反馈信息。当外设向CPU提出中断请求或DMA请求时,CPU也应有相应的响应信号反馈给外设。

2. 接口的基本组成

如上所述,接口中要分别传送数据信息、控制信息和状态信息,这些信息都是通过数据总线来传送。大多数计算机都把外部设备的状态信息视为输入数据,而把控制信息看成输出数据,并在接口中分设各自相应的寄存器,赋予不同的端口地址,各种信息分时地使用数据总线传送到各自的寄存器中。接口的基本组成及与主机、外设间的连接如图9-1所示。

接口(Interface)与端口(Port)是两个不同的概念。端口是指接口电路中可以被CPU直接访问的寄存器,若干个端口加上相应的控制逻辑电路才组成接口。

通常,一个接口中包含数据端口、命令端口和状态端口。存放数据信息的寄存器称为数据端口,存放控制命令的寄存器称为命令端口,存放状态信息的寄存器称为状态端口。CPU通过输入指令可以从有关端口中读取信息,通过输出指令可以把信息写入有关端口。

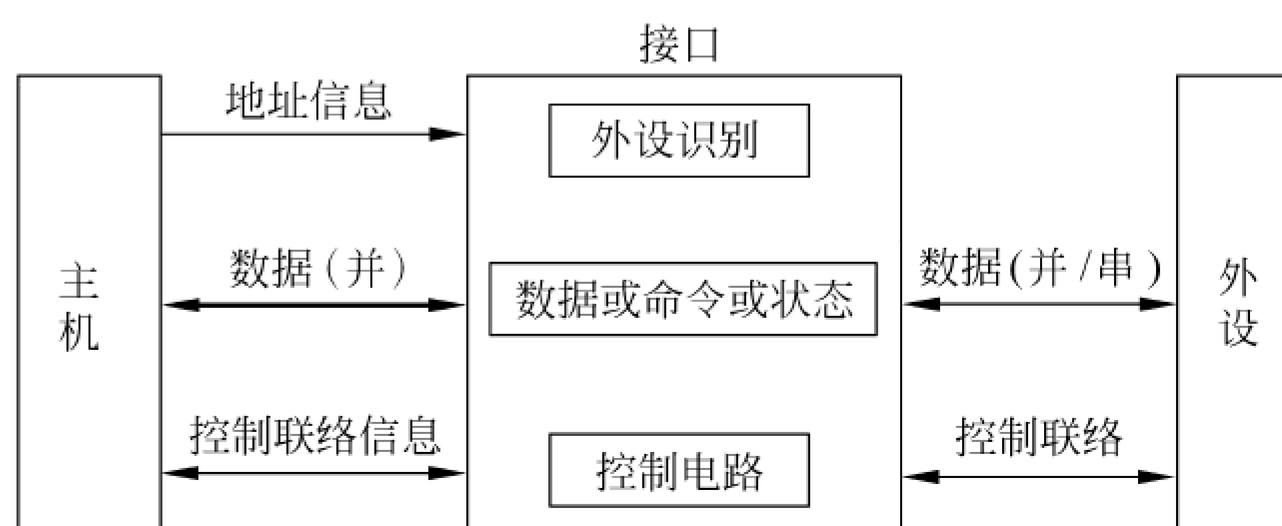


图 9-1 接口与主机、外设间的连接

CPU 对不同端口的操作有所不同,有的端口只能写或只能读,有的端口既可以读又可以写。例如,对状态端口只能读,将外设的状态标志送到 CPU 中;对命令端口只能写,将 CPU 的各种控制命令发送给外设。为了节省硬件,在有的接口电路中,状态信息和控制信息可以共用一个寄存器(端口),称为设备的控制/状态寄存器。

3. 接口的类型

输入输出接口的分类可以从不同的角度来考虑。

(1) 按数据传送方式分类

输入输出接口有串行接口和并行接口。这里所说的数据传送方式指的是外设和接口一侧的传送方式,而在主机和接口一侧数据总是并行传送的,如图 9-1 所示。在并行接口中,外设和接口间的传送宽度是一个字节(或字)的所有位,一次传输的信息量大,但数据线的数目将随着传送数据宽度的增加而增加。在串行接口中,外设和接口间的数据是一位接一位地串行传送的,一次传输的信息量小,但只需一根数据线。在远程终端和计算机网络等设备离主机较远的场合下,用串行接口比较经济合算。

(2) 按主机访问 I/O 设备的控制方式分类

输入输出接口可分为程序查询式接口、程序中断接口、DMA 接口,以及更复杂一些的通道控制器等。

(3) 按功能选择的灵活性分类

输入输出接口有可编程接口和不可编程接口。可编程接口的功能及操作方式是由程序来改变或选择的,用编程的手段可使一块接口芯片执行多种不同的功能。不可编程接口则不能由程序来改变其功能,只能用硬连线逻辑来实现不同的功能。

(4) 按通用性分类

输入输出接口有通用接口和专用接口。通用接口是可供多种外设使用的标准接口,通用性强。专用接口是为某类外设或某种用途专门设计的。

(5) 按输入输出的信号分类

输入输出接口有数字接口和模拟接口。数字接口的输入输出全为数字信号,以上列举的并行接口和串行接口都是数字接口。而模数转换器和数模转换器属于模拟接口。

(6) 按应用来分类

输入输出接口分为以下几种:

① 运行辅助接口。运行辅助接口是计算机日常工作所必需的接口器件,包括数据总

线、地址总线和控制总线的驱动器和接收器、时钟电路、磁盘接口及磁带接口。

② 用户交互接口。这类接口包括计算机终端接口、键盘接口、图形显示器接口及语音识别与合成接口等。

③ 传感接口。包括温度传感接口、压力传感接口和流量传感接口等。

④ 控制接口。这类接口用于计算机控制系统。

9.1.3 外设的识别与端口寻址

为了能在众多的外设中寻找或挑选出要与主机进行信息交换的外设,就必须对外设进行编址。外设识别是通过地址总线和接口电路中的外设识别电路来实现的,I/O 端口地址就是主机与外设直接通信的地址,CPU 可以通过端口发送命令、读取状态和传送数据。如何实现对这些端口的访问,这就是所谓的 I/O 端口的编址方式。

1. 端口地址编址方式

I/O 端口编址方式有两种:一种是 I/O 映射方式,即把 I/O 端口地址与存储器地址分别进行独立的编址;另一种是存储器映射方式,即把端口地址与存储器地址统一编址。这个问题已在第 3 章中作过介绍,这里则从外设识别的角度加以进一步讨论。

(1) 独立编址

在这种编址方式中,主存地址空间和 I/O 端口地址空间是相对独立的,分别单独编址。例如,在 8086 中,其主存地址范围是从 00000H 到 FFFFFH,其 I/O 端口地址范围是从 0000H 到 FFFFH,它们互相独立,互不影响。CPU 访问主存时,由主存读写控制线控制;访问外设时,由 I/O 读写控制线控制,所以在指令系统中必须设置专门的 I/O 指令。当 CPU 使用 I/O 指令时,其指令的地址字段直接或间接的指示出端口地址。这些端口地址被接口电路中的地址译码器接收并且进行译码,符合者就是 CPU 所指定的外设寄存器,该外设寄存器将被 CPU 访问。

(2) 统一编址

在这种编址方式中,I/O 端口地址和主存单元地址是统一编址的,把 I/O 接口中的端口作为主存单元一样进行访问,不设置专门的 I/O 指令。当 CPU 访问外设时,把分配给该外设的地址码(具体到该外设接口中的某一寄存器号)送到地址总线上,然后各外设接口中的地址译码器对地址码进行译码,如果符合即是 CPU 指定的外设寄存器。

例如,PDP-11 机分配给某些外设寄存器的端口地址如下。

纸带输入机:	控制状态寄存器	177550Q
	数据缓冲寄存器	177552Q
穿孔输入机:	控制状态寄存器	177554Q
	数据缓冲寄存器	177556Q
控制台打字机:	键盘控制状态寄存器	177560Q
	键盘数据寄存器	177562Q
	打印控制状态寄存器	177564Q
	打印数据寄存器	177566Q
行式打印机:	控制状态寄存器	177514Q

	数据缓冲寄存器	177516Q
磁盘存储器：	盘驱动器寄存器	177400Q
	错误寄存器	177402Q
	控制状态寄存器	177404Q
	字计数寄存器	177406Q
	主存地址寄存器	177410Q
	盘数据地址寄存器	177412Q
	数据缓冲寄存器	177414Q

从这个例子可以看出,每个外设至少有两个寄存器：控制状态寄存器和数据缓冲寄存器,外设寄存器的端口地址是连续的(PDP-11 按字节编址,外设寄存器字长 16 位)。在 PDP-11 中,把主存的高 4KB 地址空间留给外设寄存器和 CPU 内部寄存器使用,这 4KB 存储空间不允许用户再存放其他内容。

2. 独立编址方式的端口访问

独立编址方式广泛应用于 Intel 系列微型计算机及大型计算机中,Intel 80x86 的 I/O 地址空间由 64K(2^{16})个独立编址的 8 位端口组成。两个连续的 8 位端口可作为 16 位端口处理;4 个连续的 8 位端口可作为 32 位端口处理。

80x86 的专用 I/O 指令 IN 和 OUT 有直接寻址和间接寻址两种类型。直接寻址 I/O 端口的寻址范围为 00～FFH,至多为 256 个端口地址。这时程序可以指定：

编号 0～255 的 256 个 8 位端口;

编号 0、2、4、…、252、254 的 128 个 16 位端口;

编号 0、4、8、…、248、252 的 64 个 32 位端口。

间接寻址由 DX 寄存器间接给出 I/O 端口地址。DX 寄存器长 16 位,所以最多可寻址 $2^{16}=64K$ 个端口地址,这时程序可指定：

编号 0～65 535 的 65 536 个 8 位端口;

编号 0、2、4、…、65 532、65 534 的 32 768 个 16 位端口;

编号 0、4、8、…、65 528、65 532 的 16 384 个 32 位端口。

CPU 一次可实现字节(8 位)、字(16 位)或双字(32 位)的数据传送。32 位端口应对准可被 4 整除的地址;16 位端口应对准偶地址。

9.1.4 输入输出信息传送控制方式

主机和外设之间的信息传送控制方式,经历了由低级到高级、由简单到复杂、由集中管理到各部件分散管理的发展过程,按其发展的先后次序和主机与外设并行工作的程度,可以分为程序查询方式、程序中断方式、直接存储器存取方式和 I/O 通道控制方式 4 种。

1. 程序查询方式

程序查询方式是一种程序直接控制方式,这是主机与外设之间进行信息交换的最简单方式,输入和输出完全是通过 CPU 执行程序来完成的。一旦某一外设被选中并启动之后,主机将查询这个外设的某些状态位,看其是否准备就绪?若外设未准备就绪,主机将再次查

询;若外设已准备就绪,则执行一次I/O操作。

这种方式控制简单,但外设和主机不能同时工作,各外设之间也不能同时工作,系统效率很低,因此,仅适用于外设的数目不多、对I/O处理的实时要求不那么高、CPU的操作任务比较单一且并不很忙的情况。

2. 程序中断方式

在主机启动外设后,无须等待查询,而是继续执行原来的程序,外设在做好输入输出准备时,向主机发中断请求,主机接到请求后就暂时中止原来执行的程序,转去执行中断服务程序对外部请求进行处理,在中断处理完毕后返回原来的程序继续执行。显然,程序中断不仅适用于外部设备的输入输出操作,也适用于对外界发生的随机事件的处理。

程序中断在信息交换方式中处于最重要的地位,它不仅允许主机和外设同时并行工作,并且允许多个外设同时工作。但是,完成一次程序中断还需要许多辅助操作,当外设数目较多时,中断请求过分频繁,可能使CPU应接不暇;另外,对于一些高速外设,由于信息交换是成批的,如果处理不及时,可能会造成信息丢失,因此,它主要适用于中、低速外设。

3. 直接存储器存取(DMA)方式

DMA方式是在主存和外设之间开辟直接的数据通路,可以进行基本上不需要CPU介入的主存和外设之间的信息传送,这样不仅能保证CPU的高效率,而且能满足高速外设的需要。

DMA方式只能进行简单的数据传送操作,在数据块传送的起始和结束时还需CPU及中断系统进行预处理和后处理。

4. I/O通道控制方式

I/O通道控制方式是DMA方式的进一步发展,在系统中设有通道控制部件,每个通道挂若干外设。主机执行I/O指令启动有关通道,通道执行通道程序,完成输入输出操作。

通道是一个具有特殊功能的处理器,它能独立地执行通道程序,产生相应的控制信号,实现对外设的统一管理和外设与主存之间的数据传送。但它不是一个完全独立的处理器。它要在CPU的I/O指令指挥下才能启动、停止或改变工作状态,是从属于CPU的一个专用处理器。

一个通道执行输入输出过程全部由通道按照通道程序自行处理,不论交换信息多少,只打扰CPU两次(启动和停止时)。因此,主机、外设和通道可以并行同时工作,而且一个通道可以控制多台不同类型的设备。

目前,小型和微型计算机大多采用程序查询方式、程序中断方式和DMA方式;大型和中型计算机多采用通道方式。

9.2 程序查询方式及其接口

程序查询方式是主机与外设之间进行信息交换的最简单方式,程序查询方式的核心问题在于需要不断地查询I/O设备是否准备就绪。

9.2.1 程序查询方式

1. 程序查询的基本思想

由CPU执行一段输入输出程序来实现主机与外设之间数据传送的方式称为程序直接控制方式。根据外设的不同性质,这种传送方式又可分为无条件传送和程序查询方式两种。

在无条件传送方式中,I/O端口总是准备好接收主机的输出数据,或总是准备好向主机输入数据,因而CPU无须查询外设的工作状态,而默认外设始终处于准备就绪状态。在CPU认为需要时,随时可直接利用I/O指令访问相应的I/O端口,实现与外设之间的数据交换。这种方式的优点是软、硬件结构都很简单,但要求时序配合精确,一般的外设难以满足要求。所以,只能用于简单开关量的输入输出控制中,稍复杂一点的外设都不采用此种方式。

许多外设的工作状态是很难事先预知的,例如,何时按键、打印机是否能接收新的打印输出信息等。当CPU与外设工作不同步时,很难确保CPU在执行输入操作时,外设一定是"准备好"的;而在执行输出操作时,外设一定是"缓冲器空"的。为了保证数据传送的正确进行,就要求CPU在程序中查询外设的工作状态。如果外设尚未准备就绪,CPU就循环等待,只有当外设已做好准备,CPU才能执行I/O指令进行数据传送,这就是程序查询方式。

2. 程序查询方式的工作流程

程序查询方式的工作过程如下:

① 预置传送参数。在传送数据之前,由CPU执行一段初始化程序,预置传送参数。传送参数包括存取数据的主存缓冲区首地址和传送数据的个数。

② 向外设接口发出命令字。当CPU选中某台外设时,执行输出指令向外设接口发出命令字启动外设,为接收数据或发送数据做应有的操作准备。

③ 从外设接口取回状态字。CPU执行输入指令,从外设接口中取回状态字并进行测试,判断数据传送是否可以进行。

④ 查询外设标志。CPU不断查询状态标志。如果外设没有准备就绪,CPU就踏步等待,转至第③步,一直到这个外设准备就绪,并发出"外设准备就绪"信号为止。

⑤ 传送数据。只有外设准备好,才能实现主机与外设间的一次数据传送。输入时,CPU执行输入指令,从外设接口的数据缓冲寄存器中接收数据;输出时,CPU执行输出指令,将数据写入外设接口的数据缓冲寄存器中。

⑥ 修改传送参数。每进行一次数据传送之后必须要修改传送参数,其中包括主存缓冲区地址加1,传送个数计数器减1。

⑦ 判断传送是否结束。如果传送个数计数器不为0,则转第③步,继续传送,直到传送个数计数器为0,表示传送结束。

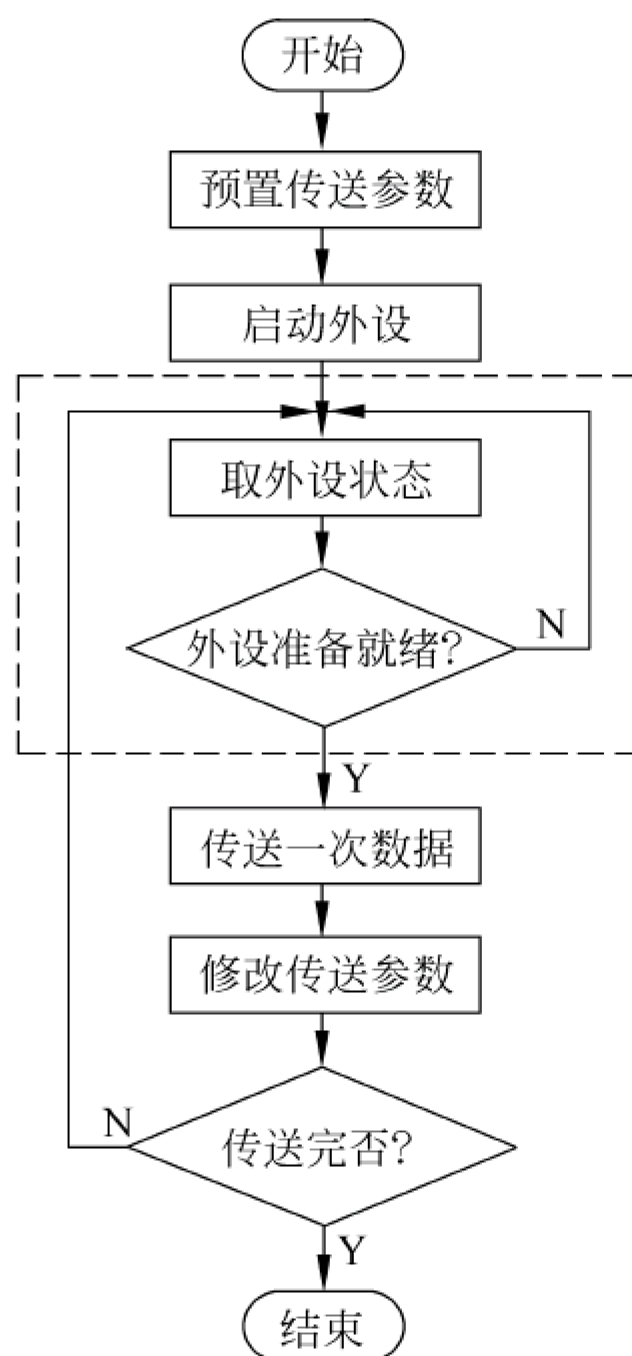


图9-2 程序查询方式流程图

程序查询方式的工作流程如图9-2所示,其程序查询的

核心如图中虚线框部分，真正传送数据的操作由输入或输出指令完成。

9.2.2 程序查询方式接口

程序查询方式是最简单、经济的I/O方式，只需很少的硬件。通常接口中至少有两个寄存器，一个是数据缓冲寄存器，即数据端口，用来存放与CPU进行传送的数据信息；另一个是供CPU查询的设备状态寄存器，即状态端口，这个寄存器由多个标志位组成，其中最重要的是“外设准备就绪”标志（输入或输出设备的准备就绪标志可以不是同一位）。当CPU得到这位标志后就进行判断，以决定下一步是继续循环等待还是进行I/O传送。也有些计算机仅设置状态标志触发器，其作用与设备状态寄存器相同。

1. 输入接口

图9-3为查询式输入接口电路，图中Ready为准备好触发器，它对应于设备状态寄存器的D_0位。

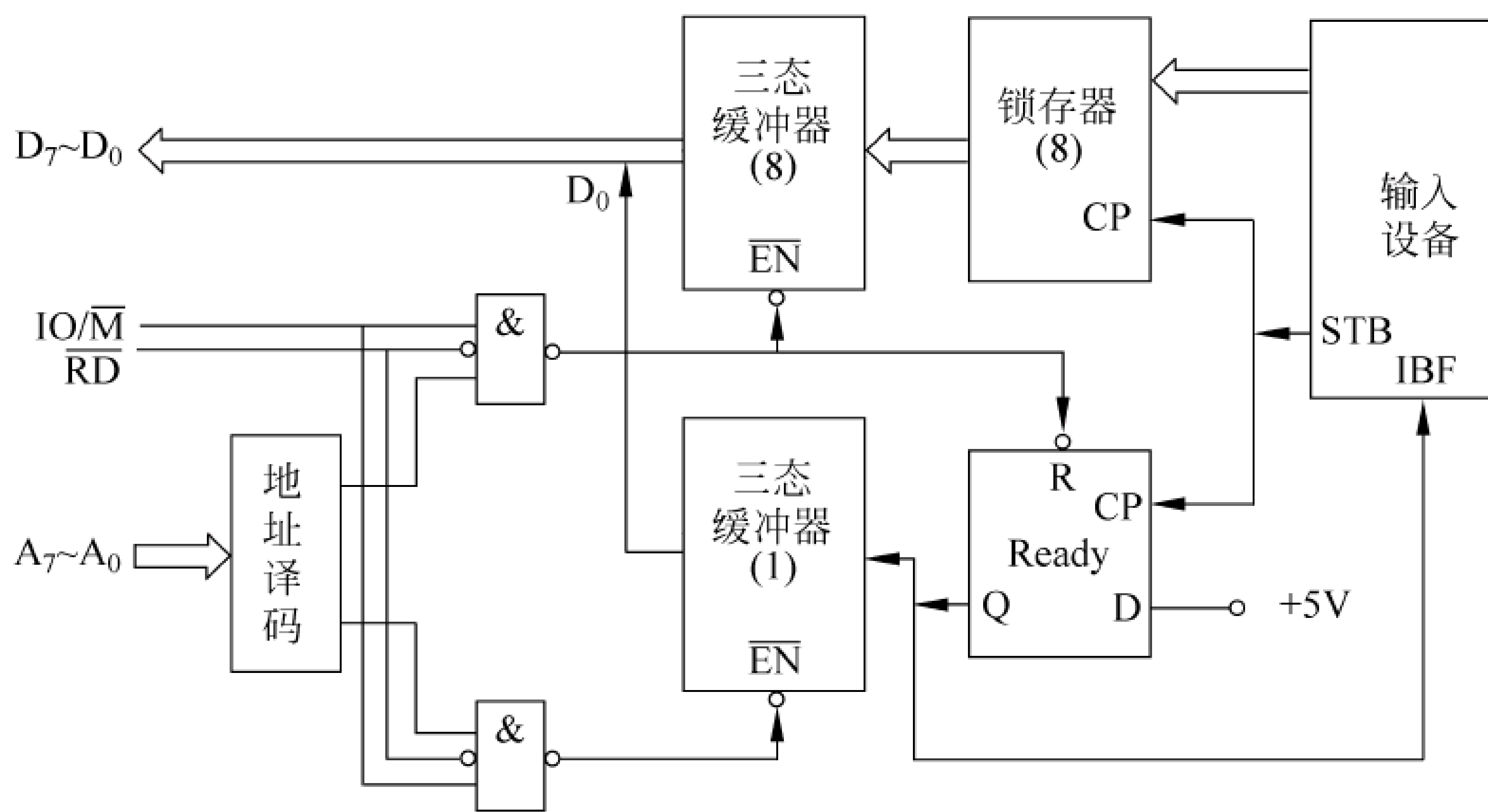


图9-3 查询式输入接口电路

在输入设备准备好数据时，发出一个选通信号（STB），将数据送入锁存器，同时将Ready触发器置“1”，以表示接口电路中已有数据（即准备就绪）。CPU要从外设输入数据时，先执行输入指令读取状态字，如Ready=1，再执行输入指令从锁存器中读取数据，同时把Ready触发器清“0”，以准备从外设接收下一个数据；如Ready=0，则踏步等待，继续读取状态字，直至Ready=1为止。

2. 输出接口

图9-4为查询式输出接口电路，图中Busy为忙触发器，对应于设备状态寄存器的D_7位。

输出时，CPU首先执行输入指令读取状态字，如果Busy=1，则表示接口的输出锁存器是满的，CPU只能踏步等待，继续读取状态字，直至Busy=0为止；如果Busy=0，则表示接口的输出锁存器是空的，允许CPU向外设发送数据。此时，CPU执行输出指令，将数据送入锁存器，并将Busy触发器置“1”。当输出设备把CPU送来的数据真正输出之后，将发出

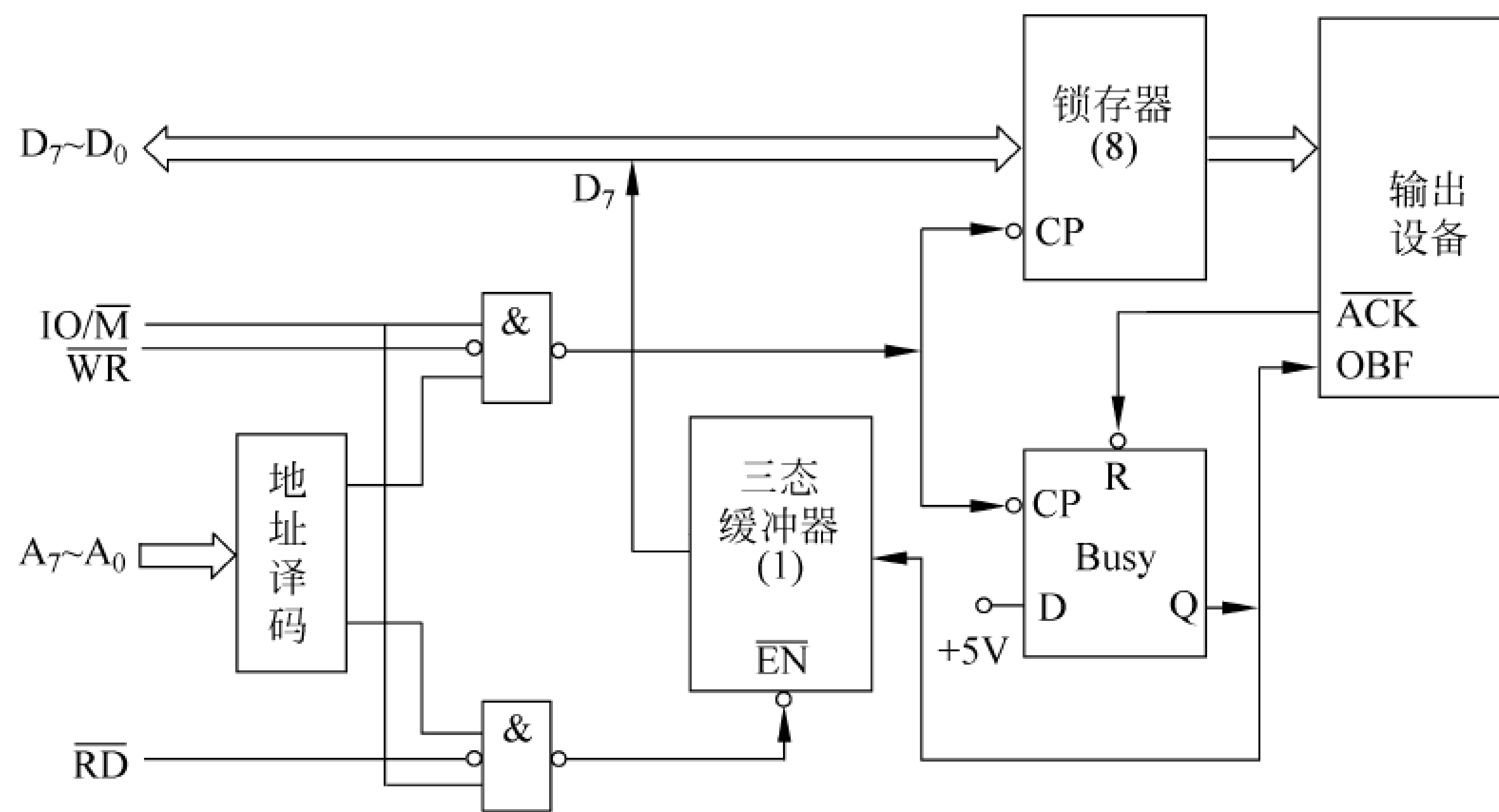


图 9-4　查询式输出接口电路

一个$\overline{ACK}$信号，使 Busy 触发器置"0"，以准备下一次传送。

若有多个外设需要用查询方式工作时，其工作流程如图 9-5 所示。此时，CPU 巡回检测各个外设，逐个进行查询，发现哪个外设准备就绪，就对该外设实施数据传送，然后再对下一外设查询，依次循环。在整个查询过程中，CPU 不能做别的事。如果某一外设刚好在查询过自己之后才处于就绪状态，那么它就必须等 CPU 查询完其他外设后再次查询自己时，才能等到 CPU 为它服务，这对于实时性要求较高的外设来说，就可能丢失数据。

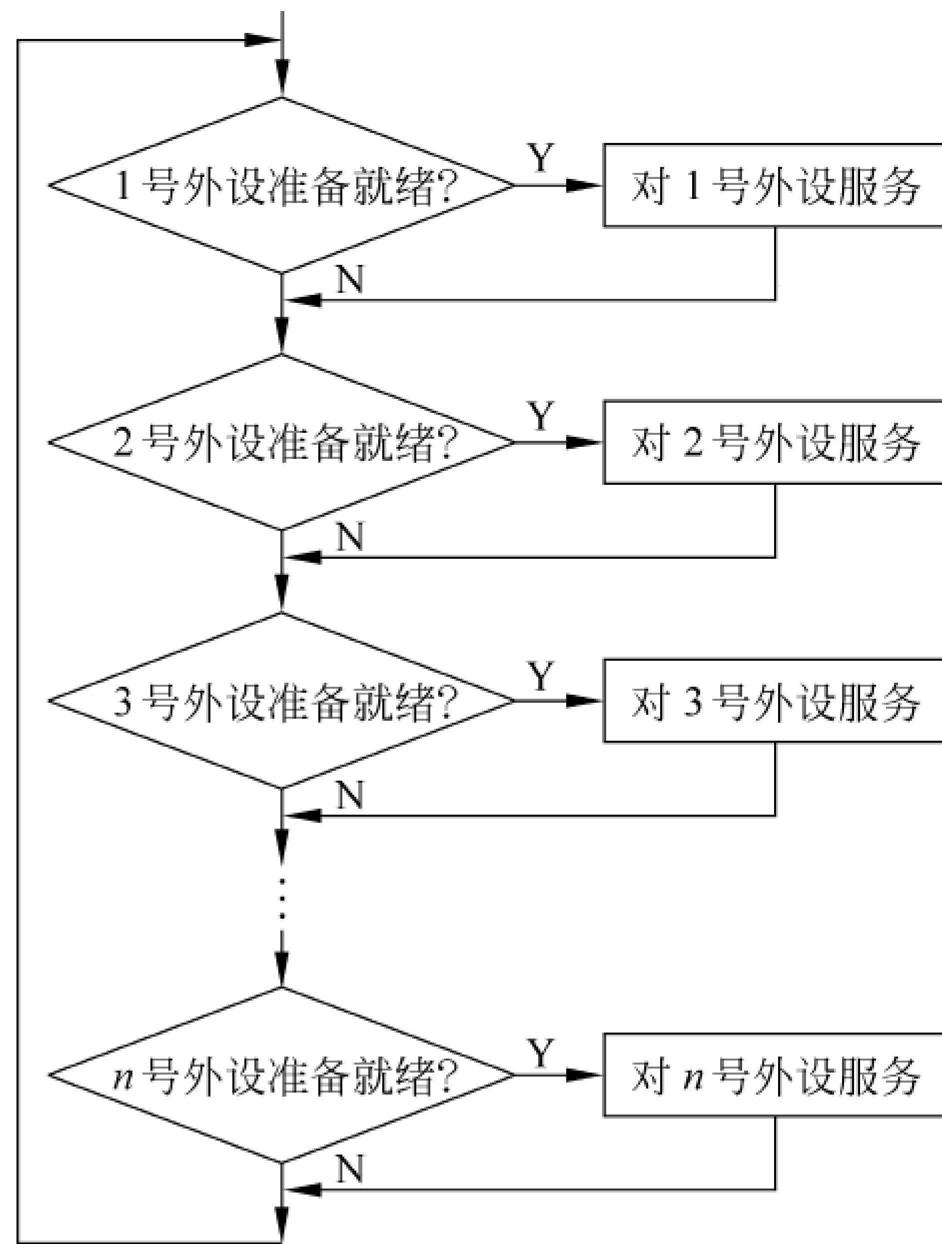


图 9-5　多个外设的查询工作流程

9.3 中断系统和程序中断方式

中断是现代计算机有效合理地发挥效能和提高效率的一个十分重要的功能。CPU中通常设有处理中断的机构——中断系统，以解决各种中断的共性问题。本节主要分析中断系统的功能，特别强调I/O中断。

9.3.1 中断的基本概念

1. 中断的提出

程序查询方式虽然简单，但却存在着下列明显的缺点：

① 在查询过程中，CPU长期处于踏步等待状态，使系统效率大大降低。

② CPU在一段时间内只能和一台外设交换信息，其他设备不能同时工作。

③ 不能发现和处理预先无法估计的错误和异常情况。

为了提高输入输出能力和CPU的效率，20世纪50年代中期，程序中断方式被引进计算机系统。程序中断方式的思想是：CPU在程序中安排好在某一时刻启动某一台外设，然后CPU继续执行原来程序，不需要像查询方式那样一直等待外设的准备就绪状态。一旦外设完成数据传送的准备工作（输入设备的数据准备好或输出设备的数据缓冲器空）时，便主动向CPU发出一个中断请求，请求CPU为自己服务。在可以响应中断的条件下，CPU暂时中止正在执行的程序，转去执行中断服务程序为中断请求者服务，在中断服务程序中完成一次主机与外设之间的数据传送，传送完成后，CPU仍返回原来的程序，从断点处继续执行。图9-6给出了程序中断方式的示意图。

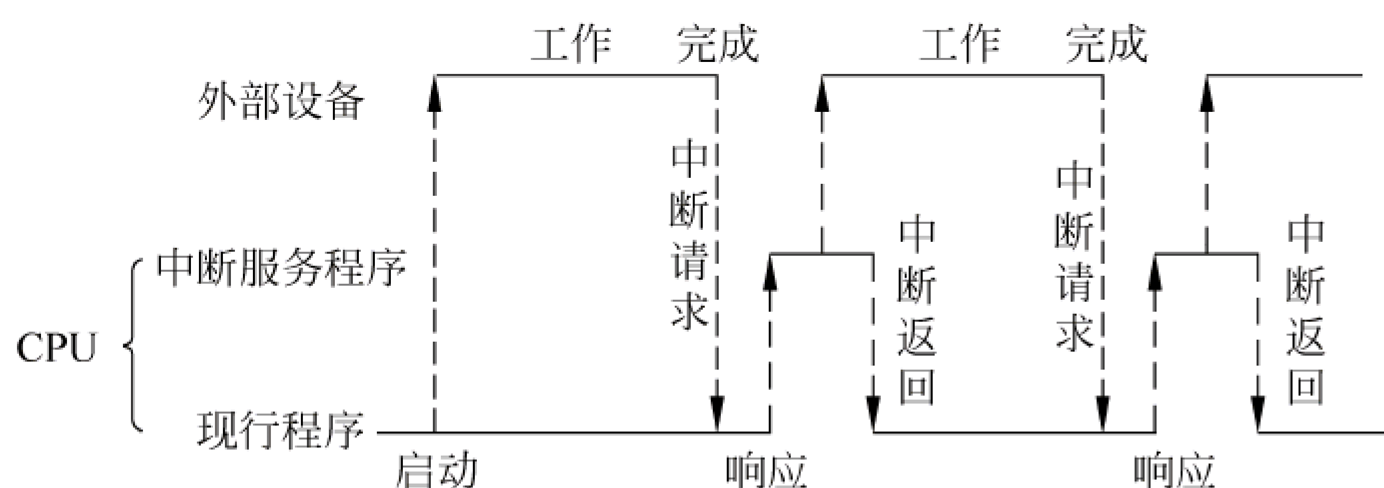


图9-6　程序中断方式

从图9-6中可以看到，中断方式在一定程度上实现了CPU和外设的并行工作，使CPU的效率得到充分的发挥。不仅如此，由于中断的引入，还能使多个外设并行工作，CPU根据需要可以启动多个外设，被启动的外设分别同时独立地工作，一旦外设准备就绪，即可向CPU发出中断请求，CPU可以根据预先安排好的优先顺序，按轻重缓急处理外设与自己的数据传送。另外，计算机在运行过程中可能会发生预料不到的异常事件，如运算错、掉电、溢出等，由于中断的引入，使计算机可以捕捉到这些故障和错误，及时予以处理。所以，现代计算机无论是巨型机、大型机、小型机还是微型机都具有中断处理的能力。

从图9-6中还可以看到，中断的处理过程实际上是程序的切换过程，即从现行程序切换到中断服务程序，再从中断服务程序返回到现行程序。CPU每次执行中断服务程序前总要

保护断点、保护现场,执行完中断服务程序返回现行程序之前又要恢复现场、恢复断点。这些中断的辅助操作都将会限制数据传送的速度。

中断系统是计算机实现中断功能的软、硬件总称。一般在 CPU 中配置中断机构,在外设接口中配置中断控制器,在软件上设计相应的中断服务程序。

2. 程序中断与调用子程序的区别

程序中断是指计算机执行现行程序的过程中,出现某些急需处理的异常情况和特殊请求,CPU 暂时中止现行程序,而转去对随机发生的更紧迫的事件进行处理,在处理完毕后,CPU 将自动返回原来的程序继续执行。

从表面上看起来,计算机的中断处理过程有点类似于调用子程序的过程,这里现行程序相当于主程序,中断服务程序相当于子程序。但是,它们之间却有着本质上的区别,主要的区别在于:

① 子程序的执行是由程序员事先安排好的(由一条调用子程序指令转入),而中断服务程序的执行则是由随机的中断事件引起的。

② 子程序的执行受到主程序或上层子程序的控制,而中断服务程序一般与被中断的现行程序毫无关系。

③ 不存在同时调用多个子程序的情况,而有可能发生多个外设同时请求 CPU 为自己服务的情况。

因此,中断的处理要比调用子程序指令的执行复杂得多。

3. 中断的基本类型

(1) 自愿中断和强迫中断

自愿中断又称程序自中断,它不是随机产生的中断,而是在程序中安排的有关指令,这些指令可以使机器进入中断处理的过程,如 80x86 指令系统中的软中断指令 INT n。

强迫中断是随机产生的中断,不是程序中事先安排好的。当这种中断产生后,由中断系统强迫计算机中止现行程序并转入中断服务程序。

(2) 程序中断和简单中断

程序中断就是前面提到的中断,主机在响应中断请求后,通过执行一段中断服务程序来处理更紧迫的任务,这样的中断处理过程将在后面详细讨论,它需要占用一定的 CPU 时间。

简单中断就是外设与主存之间直接进行信息交换的方法,即 DMA 方式。这种中断不去执行中断服务程序,故不破坏现行程序的状态。主机发现有简单中断请求(即 DMA 请求)时,让出一个或几个存取周期供外设与主存交换信息,然后继续执行程序。简单中断是早期对 DMA 方式的一种叫法,为避免误解,现在一般很少使用这个名词。

(3) 内中断和外中断

内中断是指由于 CPU 内部硬件或软件原因引起的中断,如单步中断、溢出中断等。

外中断是指 CPU 以外的部件引起的中断。通常,外中断又可以分为不可屏蔽中断和可屏蔽中断两种。不可屏蔽中断优先级别较高,常用于应急处理,如掉电、主存读写校验错等;而可屏蔽中断级别较低,常用于一般 I/O 设备的数据传送。

(4) 向量中断和非向量中断

向量中断是指那些中断服务程序的入口地址是由中断事件自己提供的中断。中断事件在提出中断请求的同时,通过硬件向主机提供中断服务程序入口地址,即向量地址。

非向量中断的中断事件不能直接提供中断服务程序的入口地址。

(5) 单重中断和多重中断

单重中断在CPU执行中断服务程序的过程中不能再被打断。

多重中断在执行某个中断服务程序的过程中,CPU可去响应级别更高的中断请求,又称为中断嵌套。多重中断表征计算机中断功能的强弱,有的计算机能实现8级以上的多重中断。

9.3.2 中断请求和中断判优

1. 中断源和中断请求信号

中断源是指中断请求的来源,即引起计算机中断的事件。通常,一台计算机允许有多个中断源。由于每个中断源向CPU发出中断请求的时间是随机的,为了记录中断事件并区分不同的中断源,可采用具有存储功能的触发器来记录中断源,这个触发器称为中断请求触发器(INTR)。当某一个中断源有中断请求时,其相应的中断请求触发器置成"1"状态,表示该中断源向CPU提出中断请求。

中断请求触发器可以分散在各个中断源中,也可以集中到中断接口电路中。在中断接口电路中,多个中断请求触发器构成一个中断请求寄存器。中断请求寄存器的每一位对应一个中断源,其内容称为中断字或中断码。中断字中为"1"的位就表示对应的中断源有中断请求。

2. 中断请求信号的传送

(1) 独立请求线

每个中断源单独设置中断请求线,将中断请求信号直接送往CPU,如图9-7(a)所示。这种方式的特点是CPU在接到中断请求的同时也就知道了中断源是谁,其中断服务程序的入口地址在哪里。这有利于实现向量中断,提高中断的响应速度;但是其硬件代价较大,且CPU所能连接的中断请求线的数目有限,难以扩充。

(2) 公共请求线

多个中断源共有一根公共请求线,如图9-7(b)所示。这种方式的特点是在负载允许的情况下,中断源的数目可随意扩充;但CPU在接到中断请求后,必须通过软件或硬件的方法来识别中断源,然后再找出中断服务程序的入口地址。

(3) 二维结构

将中断请求线连成二维结构,如图9-7(c)所示。同一优先级别的中断源,采用一根公共的请求线;不同请求线上的中断源优先级别不同。这种方式综合了前两种方式的优点,在中断源较多的系统中常采用这种方式。

3. 中断优先级与判优方法

当多个中断源同时发出中断请求时,CPU在任何瞬间只能接受一个中断源的请求。究

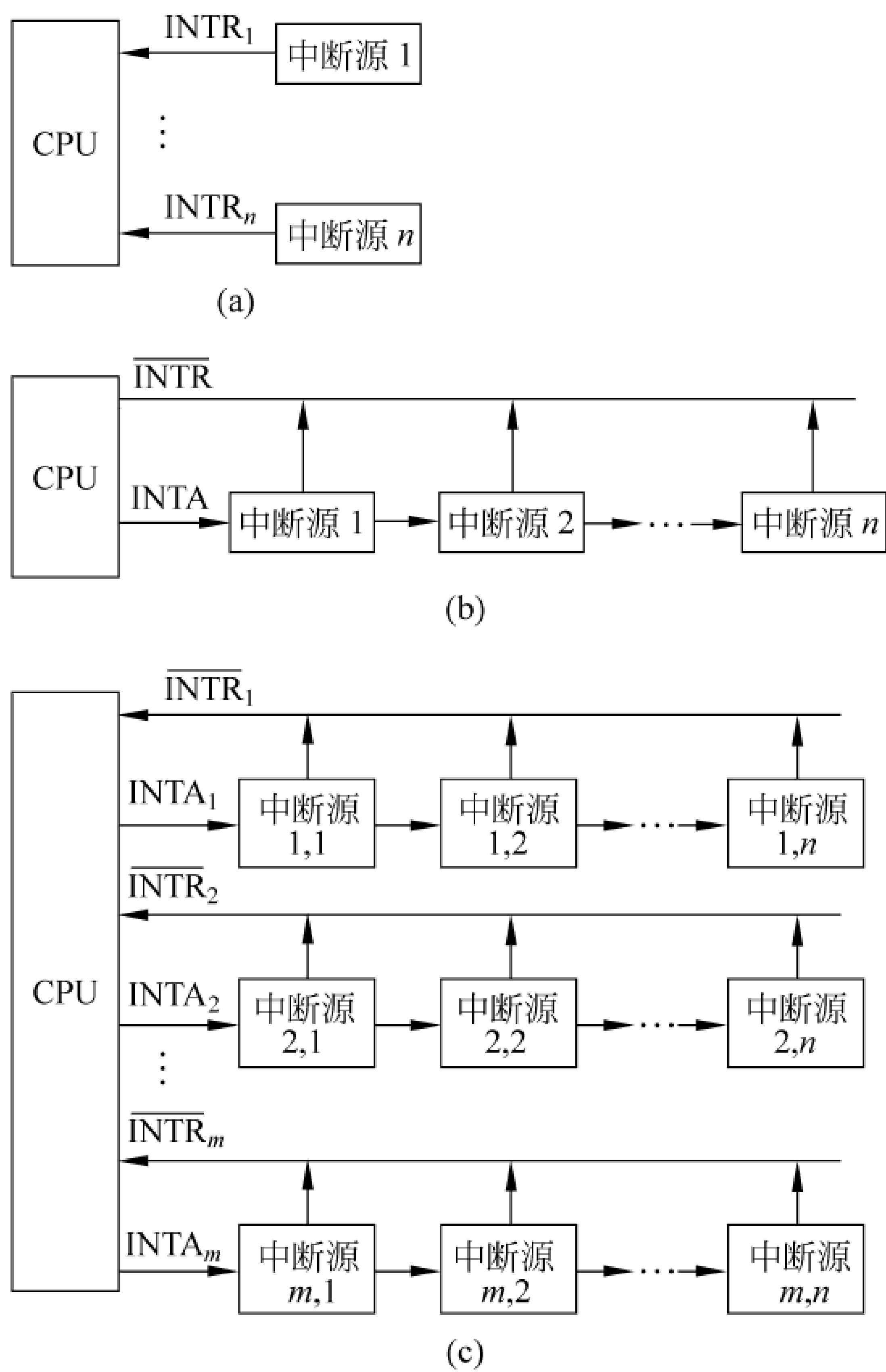


图 9-7　中断请求信号的传送

竟首先响应哪一个中断请求呢？通常，把全部中断源按中断的性质和处理的轻重缓急安排优先级，并进行排队。

确定中断优先级的原则是：对那些提出中断请求后需要立刻处理，否则就会造成严重后果的中断源规定为最高的优先级；而对那些可以延迟响应和处理的中断源规定为较低的优先级。例如，故障中断一般优先级较高，其次是简单中断，接着才是 I/O 设备中断。

每个中断源均有一个为其服务的中断服务程序，每个中断服务程序都有与之对应的优先级别。另外，CPU 正在执行的程序也有优先级。只有当某个中断源的优先级别高于 CPU 现在的优先级时，才能中止 CPU 执行当前的程序。在一些计算机的程序状态字寄存器中就设置了优先级字段，如 PDP-11 机。

中断判优的方法可分为软件判优和硬件判优两种。

(1) 软件判优方法

所谓软件判优方法，就是用程序来判别优先级，这是最简单的中断判优方法。图 9-8 给出了软件判优的流程图。当 CPU 接到中断请求信号后，就执行查询程序，逐个检测中断请求寄存器的各位状态。检测顺序是按优先级的大小排列的，最先检测的中断源具有最高的优先级，其次检测的中断源具有次高优先级，如此下去，最后检测的中断源具有最低的优先级。

显然,软件判优是与识别中断源结合在一起的,当查询到中断请求信号的发出者,也就是找到了中断源,可以立即转入对应的中断服务程序中去。

软件判优方法简单,可以灵活地修改中断源的优先级别;但查询、判优完全是靠程序实现的,不但占用 CPU 时间,而且判优速度慢。

(2) 硬件判优电路

采用硬件判优电路实现中断优先级的判定可节省 CPU 时间,判优速度快,但是成本较高。

根据中断请求信号的传送方式不同,有不同的优先排队电路,常见的方案有:独立请求线的优先排队电路、公共请求线的优先排队电路等。这些排队电路的共同特点是:优先级别高的中断请求将自动封锁优先级别低的中断请求。硬件排队电路一旦设计连接好之后,将无法改变其优先级别。

独立请求线的优先排队电路如图 9-9 所示,图中 $INTR_i'$为来自中断源的中断请求信号,$INTR_i$为经过优先排队电路后送给 CPU 的中断请求信号。

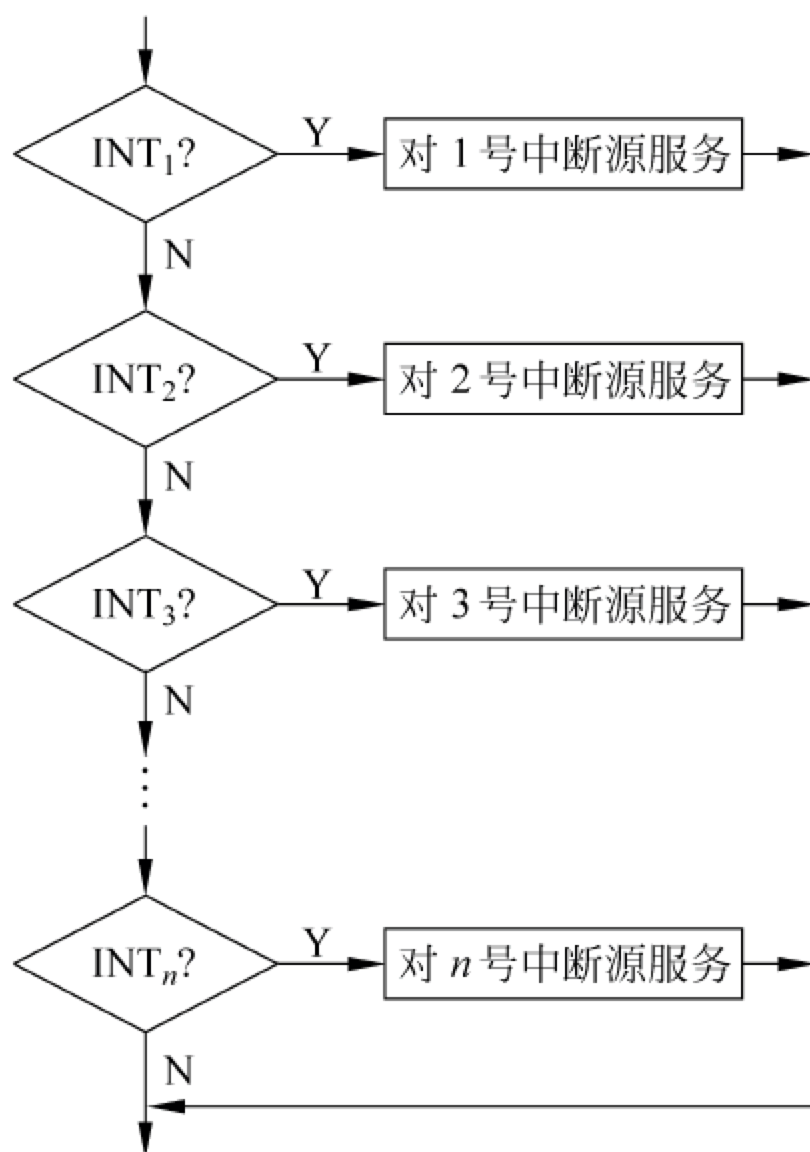


图 9-8　软件判优的流程图

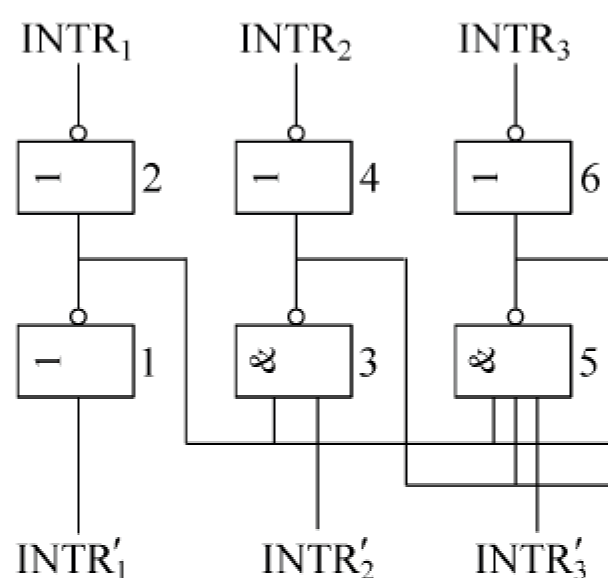


图 9-9　独立请求线的优先排队电路

优先级别从高到低依次是 $INTR_1$、$INTR_2$、$INTR_3$…优先级别高的中断请求将自动封锁优先级别低的中断请求。若 $INTR_1'=INTR_2'=1$ 时,门 1 输出的低电平将自动封锁门 3、门 5……故仅有 $INTR_1=1$,其他的 $INTR_i$均等于 0。

公共请求线的优先排队电路如图 9-10 所示,图中下面的虚线部分是一个串行优先链。$INTR_i$是各中断源的中断请求信号,优先级别从高到低依次是 $INTR_1$、$INTR_2$、$INTR_3$。而 $INTP_1$、$INTP_2$、$INTP_3$是与之对应的中断排队选中信号。$\overline{INTI}$为中断排队输入,$\overline{INTO}$为中断排队输出。若没有更高优先级的中断请求($\overline{INTI}=0$)时,$INTP_1=1$,此时如果中断请求 $INTR_1=1$,当 CPU 发来中断响应信号 INTA 时,$INTR_1$的请求被选中,选中信号经门 7 送入编码电路,产生一个唯一对应的向量地址。另一方面,由于此时$\overline{INTR_1}=0$,封锁门 2,使 $INTP_2$、$INTP_3$全为低电平,即排队识别工作不再向下进行。

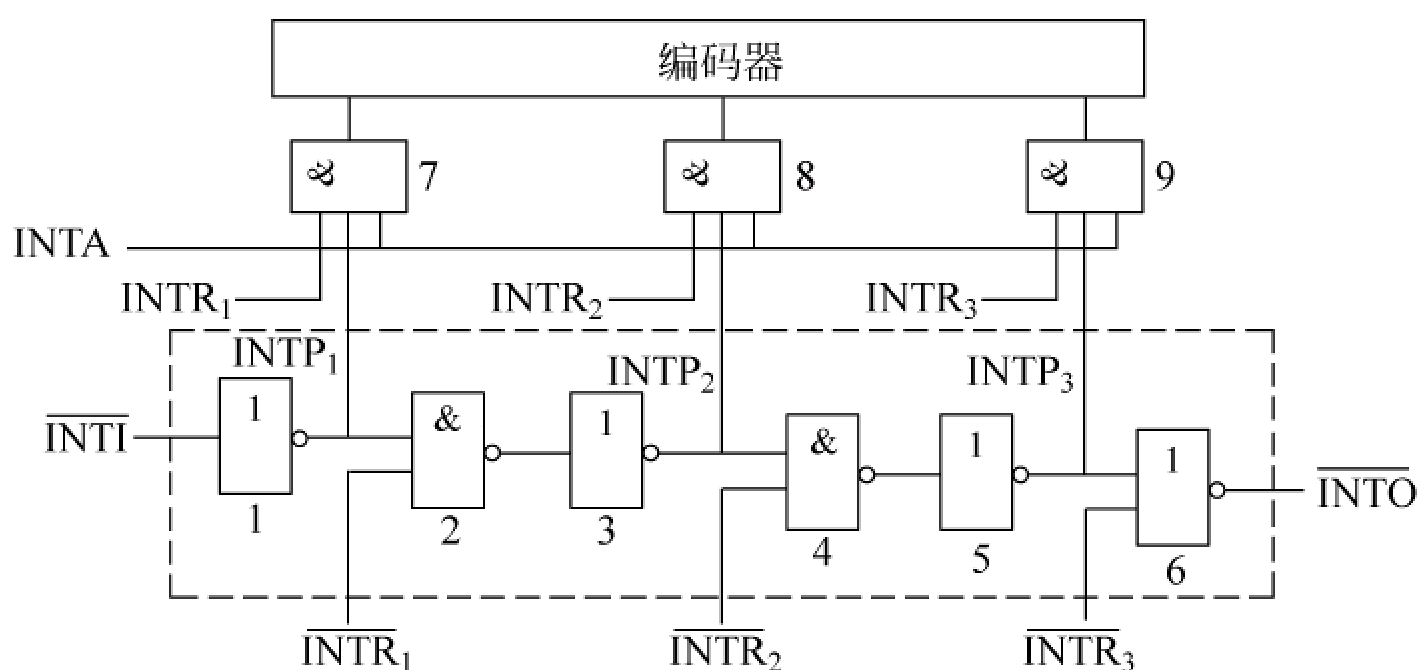


图 9-10 公共请求线的优先链排队电路

若 $INTR_1$ 无请求,则 $INTR_1=0$,门 7 被封锁,不会向编码电路送入选中信号;与此同时,因 $\overline{INTR_1}=1$,经门 2 和门 3,使 $INTP_2=1$,如果此时 $INTR_2=1$,则 $INTR_2$ 被选中。否则,串行优先链继续向下查询,直至找到发出中断请求信号 $INTR_i$ 的中断源为止。

9.3.3 中断响应和中断处理

1. CPU 响应中断的条件

CPU 响应中断必须满足下列条件。

(1) CPU 接收到中断请求信号

首先中断源要发出中断请求,同时 CPU 还要接收到这个中断请求信号。

(2) CPU 允许中断

CPU 允许中断,即开中断。CPU 内部有一个中断允许触发器(EINT),只有当 EINT=1 时,CPU 才可以响应中断源的中断请求(中断允许);如果 EINT=0,CPU 处于不允许中断状态,即使中断源有中断请求,CPU 也不响应(中断关闭)。

通常,中断允许触发器由开中断指令来置位,由关中断指令或硬件自动使其复位。

(3) 一条指令执行完毕

这是 CPU 响应中断请求的时间限制条件。一般情况下,CPU 在一条指令执行完毕且没有更紧迫的任务时才能响应中断请求。

2. 中断隐指令

CPU 响应中断之后,经过某些操作,转去执行中断服务程序。这些操作是由硬件直接实现的,把它称为中断隐指令。中断隐指令并不是指令系统中的一条真正的指令,它没有操作码,所以中断隐指令是一种不允许而且也不可能为用户使用的特殊指令。中断隐指令主要完成以下操作。

(1) 保存断点

为了保证在中断服务程序执行完毕能正确返回原来的程序,必须将原来程序的断点(即程序计数器(PC)的内容)保存起来。断点可以压入堆栈,也可以存入主存的特定单元中。

(2) 暂不允许中断

暂不允许中断即关中断。在中断服务程序中,为了保护中断现场(即 CPU 主要寄存器

的内容)期间不被新的中断所打断,必须要关中断,从而保证被中断的程序在中断服务程序执行完毕之后能接着正确地执行下去。

并不是所有的计算机都在中断隐指令中由硬件自动地关中断,也有些计算机的这一操作是由软件(中断服务程序)来实现的。

(3) 引出中断服务程序

引出中断服务程序的实质就是取出中断服务程序的入口地址送程序计数器(PC)。对于向量中断和非向量中断,引出中断服务程序的方法是不相同的。

3. 中断周期

以上几个基本操作在不同的计算机系统中的处理方法是各异的。通常,在组合逻辑控制的计算机中,专门设置了一个中断周期来完成中断隐指令的任务。在微程序控制的计算机中,则专门安排有一段微程序来完成中断隐指令的这些操作。

假设将断点存至主存的0号单元,且采用硬件向量中断法寻找中断服务程序的入口地址(向量地址=中断服务程序的入口地址),则在中断周期需完成如下操作:

① 将特定地址"0"送至存储器地址寄存器,记作 0→MAR。

② 将 PC 的内容(断点)送至 MDR,记作(PC)→MDR。

③ 向主存发写命令,启动存储器进行写操作,记作 Write。

④ 将 MDR 的内容通过数据总线写入 MAR 所指示的主存单元(0#)中,记作 MDR→M(MAR)。

⑤ 向量地址形成部件的输出送至 PC,为进入中断服务程序作准备,记作向量地址→PC。

⑥ 关中断,将中断允许触发器清 0,记作 0→EINT。

如果断点存入堆栈,只须将上述①改为堆栈指针的内容送 MAR,记作(SP)→MAR。当然断点进栈,同时需要修改栈指针。

4. 进入中断服务程序

识别中断源的目的在于使 CPU 转入为该中断源专门设置的中断服务程序。解决这个问题的方法可以用软件,也可以用硬件,或用两者相结合的方法。

软件的方法前面已经提到,由中断隐指令控制进入一个中断总服务程序,在那里判优、寻找中断源并且转入相应的中断服务程序。这种方法方便、灵活,硬件极简单,但效率较低。

下面着重讨论硬件向量中断法。当 CPU 响应某一中断请求时,硬件能自动形成并找出与该中断源对应的中断服务程序的入口地址。

向量中断的过程如图 9-11 所示。当中断源向 CPU 发出中断请求信号 $\overline{\text{INTR}}$ 之后,CPU 进行一定的判优处理。若决定响应这个中断请求,则向中断源发出中断响应信号 INTA。中断源接到 INTA 信号后就通过自己的向量地址形成部件向 CPU 发送向量地址,CPU 接收该向量地址之后就

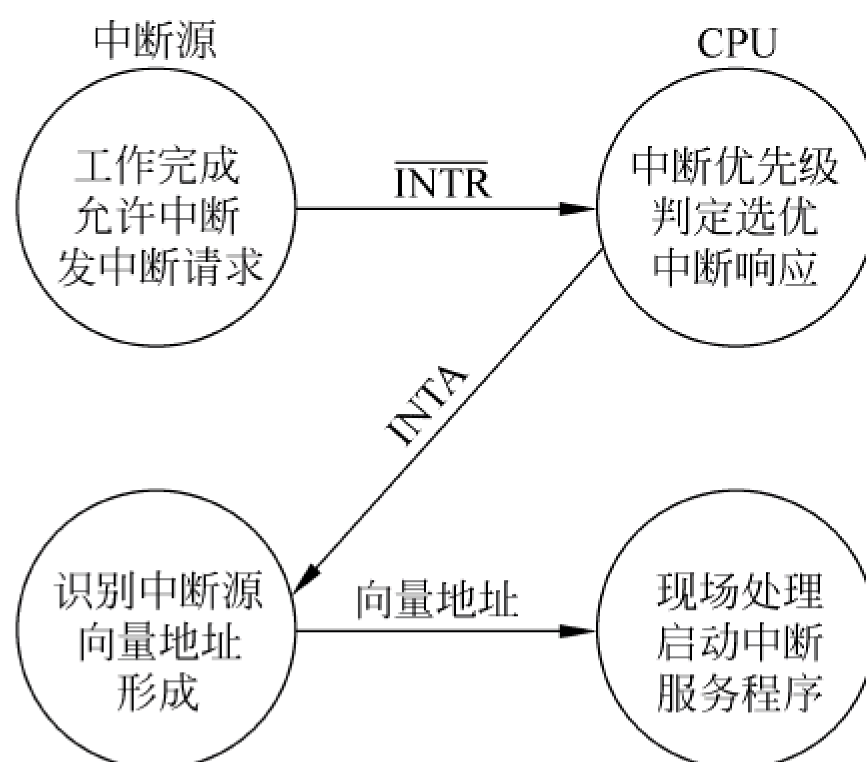


图 9-11 向量中断过程

可转入相应的中断服务程序。

向量地址通常有以下两种情况。

(1) 向量地址是中断服务程序的入口地址

如果向量地址就是中断服务程序的入口地址,则CPU不需要再经过处理就可以进入相应的中断服务程序,Z-80的中断方式0就是这种情况。各中断源在接口中由硬件电路形成一条含有中断服务程序入口地址的特殊指令(重新启动指令),从而转入相应的中断服务程序。中断源向CPU提供RST指令,其操作码为11NNN111,其中NNN为3位二进制码,范围为000～111,故RST指令有8种组合。

RST指令完成的功能如下:

① 将断点(PC的内容)压入堆栈保存。

② PC←8×NNN(向量地址)。

由此可见,RST指令能调用位于存储器前64个字节的8个中断服务程序中的任意一个,两个入口地址之间相隔有8个单元,它们依次是00H,08H,10H,…,38H。如果中断服务程序较短,就可以放在这些单元里;如果中断服务程序较长,可在这8个单元里再放一条转移指令,以转至真正的中断服务程序中去。例如,当指令为RST 7时,经CPU处理后得到的向量地址VA=0038H,即该中断源的中断服务程序的入口地址为0038H。

(2) 向量地址是中断向量表的指针

如果向量地址是中断向量表的指针,则向量地址指向一个中断向量表,从中断向量表的相应单元中再取出中断服务程序的入口地址,此时中断源给出的向量地址是中断服务程序入口地址的地址。目前,大多数微型计算机都采用这种方法,Intel 8086和Z-80的中断方式2都属于这种情况,其转中断服务程序的方法如图9-12所示。

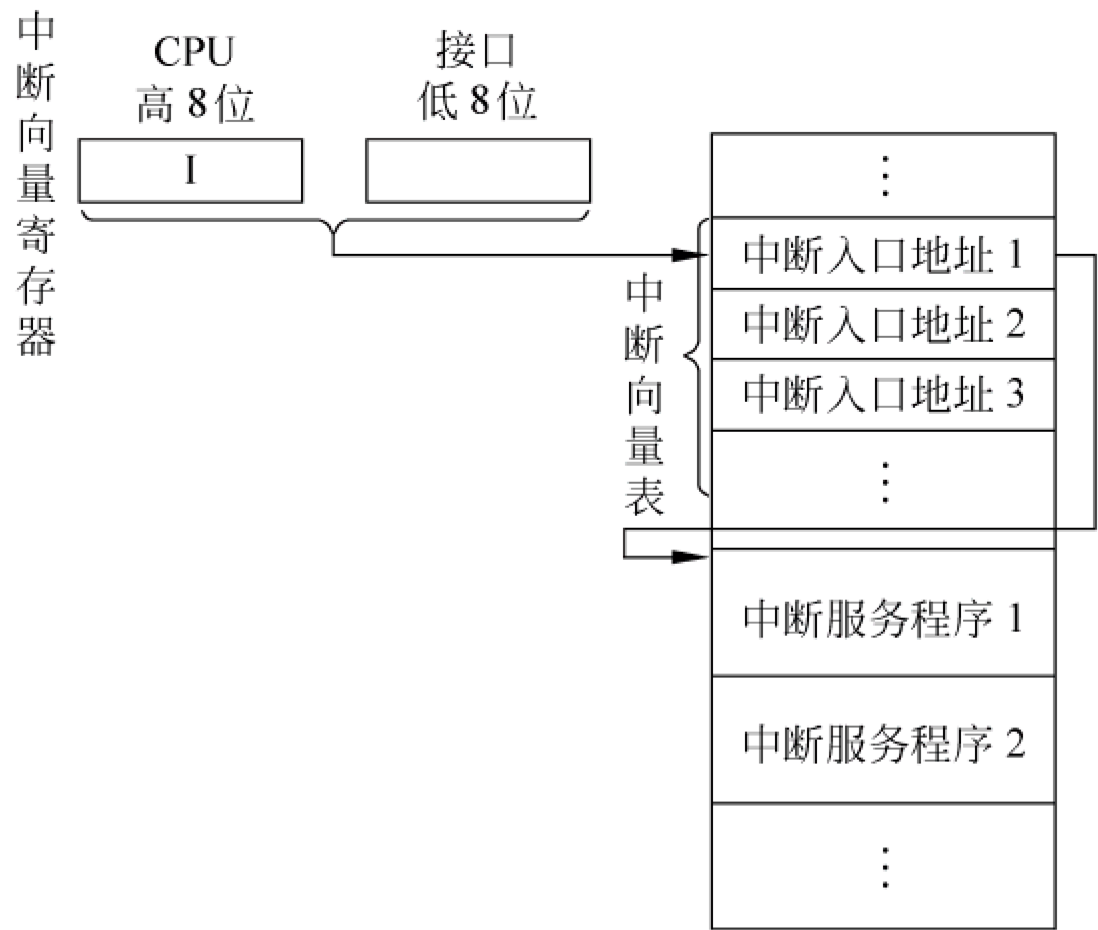


图9-12 Z-80的中断方式2

5. 中断现场的保护和恢复

中断现场指的是发生中断时CPU的主要状态,其中最重要的是断点,另外还有一些通

用寄存器的状态。之所以需要保护和恢复现场的原因是因为CPU要先后执行两个完全不同的程序(现行程序和中断服务程序),必须进行两种程序运行状态的转换。一般来说,在中断隐指令中,CPU硬件将自动保存断点,有些计算机还自动保存程序状态字寄存器(PSWR)的内容。但是,在许多应用中,要保证中断返回后原来的程序能正确地继续运行,仅保存这一两个寄存器的内容是不够的。为此,在中断服务程序开始时,应由软件去保存那些硬件没有保存,而在中断服务程序中又可能用到的寄存器(如某些通用寄存器)的内容,在中断返回之前,这些内容还应该被恢复。

现场的保护和恢复方法不外乎有纯软件和软、硬件相结合两种。纯软件方法是在CPU响应中断后,用一系列传送指令把要保存的现场参数传送到主存某些单元中,当中断服务程序结束后,再采用传送指令进行相反方向的传送。这种方法不需要硬件代价,但是占用了CPU的宝贵时间,速度较慢。现代计算机一般都先采用硬件方法来自动快速的保护和恢复部分重要的现场,其余寄存器的内容再由软件完成保护和恢复,这种方法的硬件支持是堆栈。

软、硬件保护现场往往是和向量中断结合在一起使用的。首先把断点和程序状态字自动压入堆栈,这就是保护旧现场;接着根据中断源送来的向量地址自动取出中断服务程序入口地址和新的程序状态字,这就是建立新现场;最后由一些指令实现对必要的通用寄存器的保护。恢复现场则是保护现场的逆处理。

9.3.4 多重中断与中断屏蔽

1. 中断嵌套

中断嵌套过程如图9-13所示。中断嵌套的层次可以有多层,越在里层的中断请求越急迫,优先级越高,因此优先得到CPU的服务。

要使计算机具有多重中断的能力,首先要能保护多个断点,而且先发生的中断请求的断点,先保护后恢复;后发生的中断请求的断点,后保护先恢复。堆栈的先进后出特点正好满足多重中断这一先后次序的需要。同时,在CPU进入某一中断服务程序之后,系统必须处于开中断状态,否则中断嵌套是不可能实现的。

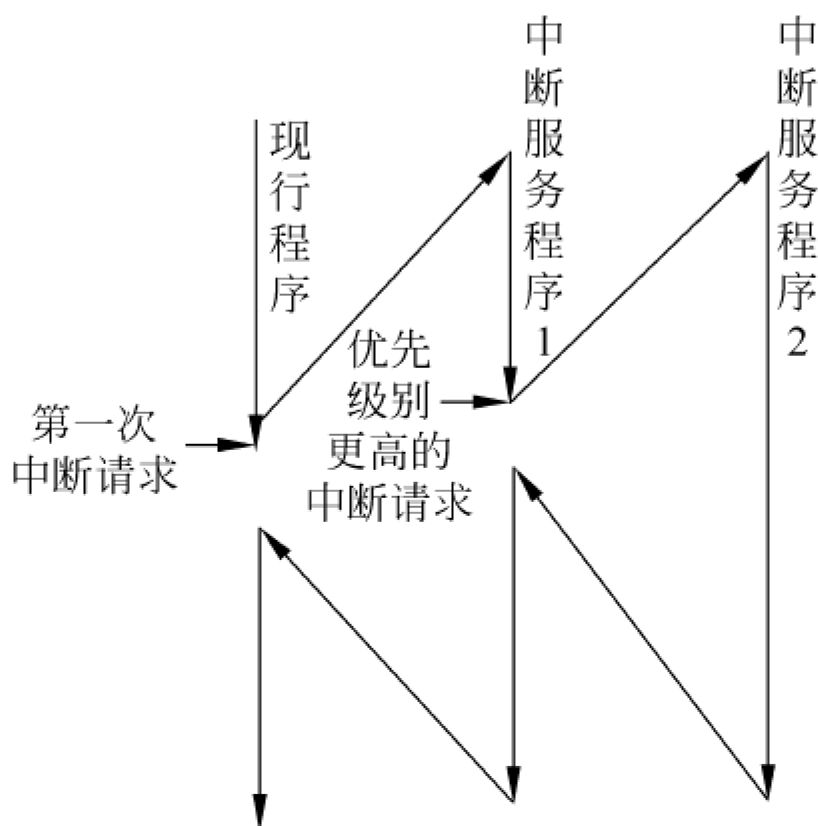


图9-13 中断嵌套

2. 允许和禁止中断

允许中断还是禁止中断是用CPU中的中断允许触发器控制的,当中断允许触发器(EINT)被置"1",则允许中断;当中断允许触发器(EINT)被置"0",则禁止中断。

允许中断即开中断,下列情况应开中断:

① 无论是单重中断还是多重中断,在中断服务程序执行完毕,恢复中断现场之后。

② 在多重中断的情况下,保护中断现场之后。

禁止中断即关中断,下列情况应关中断:

① 当响应某一级中断请求,不再允许被其他中断请求打断时。

② 在中断服务程序的保护和恢复现场之前。

3. 中断屏蔽

中断源发出中断请求之后,这个中断请求并不一定能真正送入CPU,在有些情况下,可以用程序方式有选择地封锁部分中断,这就是中断屏蔽。

如果给每个中断源都相应地配备一个中断屏蔽触发器(MASK),则每个中断请求信号在送往判优电路之前,还要受到屏蔽触发器的控制。当MASK=1,表示对应中断源的请求被屏蔽,可见中断请求触发器和中断屏蔽触发器是成对出现的,只有当$INTR_i=1$(中断源有中断请求),$MASK_i=0$(该级中断未被屏蔽)时,才允许对应的中断请求送往CPU,相应的电路如图9-14所示。

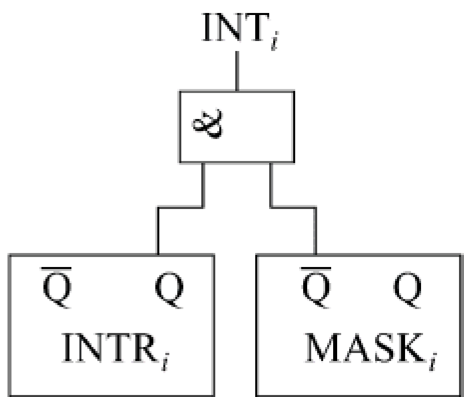


图9-14 中断请求触发器和中断屏蔽触发器

在中断接口电路中,多个屏蔽触发器组成一个屏蔽寄存器,其内容称为屏蔽字或屏蔽码,由程序来设置。屏蔽字某一位的状态将成为本中断源能否真正发出中断请求信号的必要条件之一。这样,就可实现CPU对中断处理的控制,使中断能在系统中合理协调地进行。中断屏蔽寄存器的作用如图9-15所示。具体地说,用程序设置的方法将屏蔽寄存器中的某一位置“1”,则对应的中断请求被封锁,无法去参加排队判优;若屏蔽寄存器中的某一位置“0”,才允许对应的中断请求送往CPU。

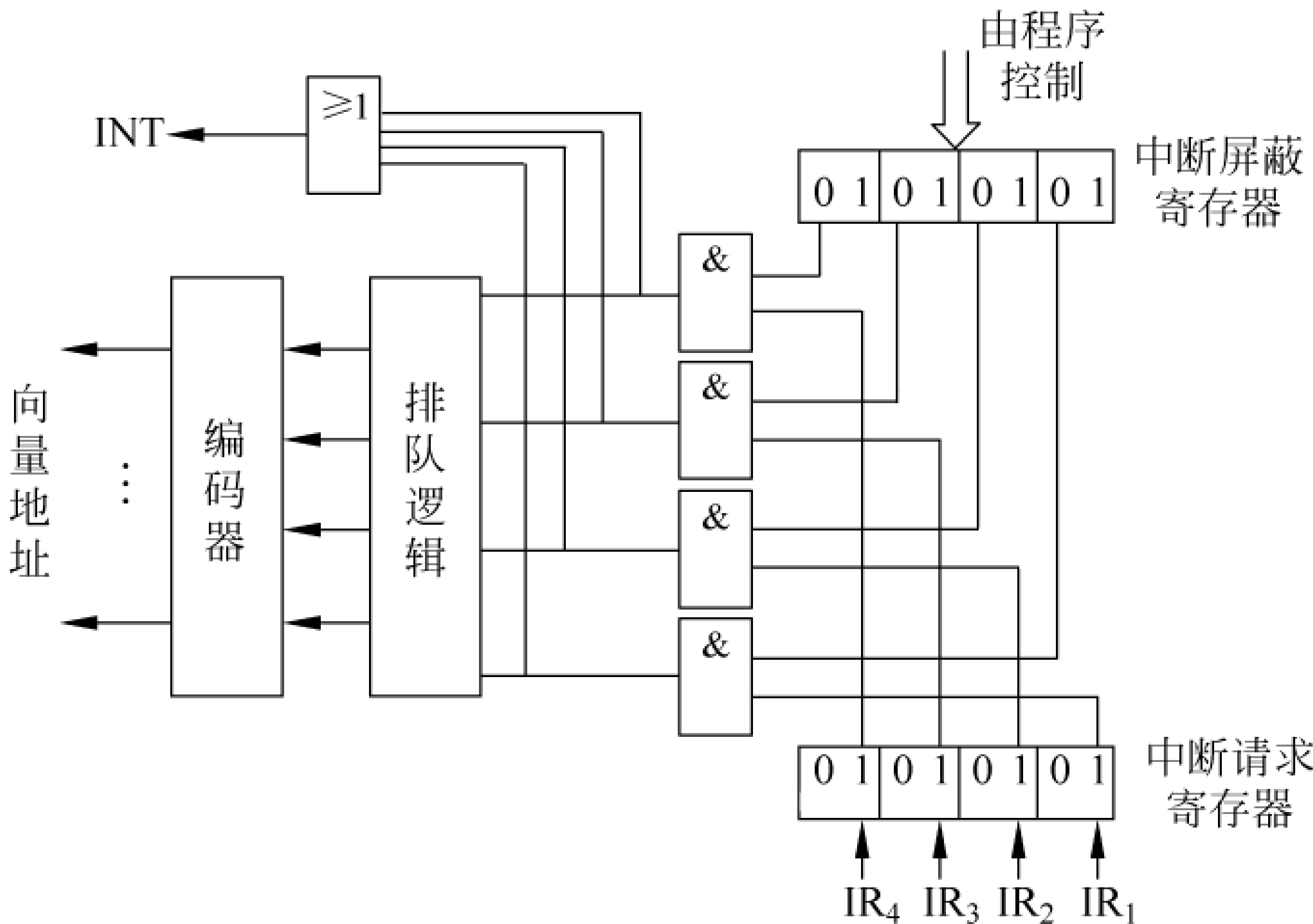


图9-15 中断屏蔽寄存器的作用

例如,一个中断系统有16个中断源,每一个中断源按其优先级别赋予一个屏蔽字。屏蔽字与中断源的优先级别是一一对应的,“0”表示开放,“1”表示屏蔽。表9-1中列出了各中断源对应的屏蔽字。

表 9-1 各中断源的屏蔽字

中断源的优先级	屏蔽字(16 位)
1	111…111
2	011…111
3	001…111
⋮	⋮
15	000…011
16	000…001

表 9-1 中第 1 级中断源的屏蔽字是 16 个“1”,它的优先级别最高,禁止本级和更低级的中断请求……第 16 级中断源的屏蔽字只有第 16 位(最低位)为“1”,其余各位均为“0”,它的优先级别最低,仅禁止本级的中断请求,而对其他高级的中断请求全部开放。

也有些中断请求是不可屏蔽的,即不受中断屏蔽寄存器的控制。这种中断源的中断请求一旦提出,CPU 必须立即响应,它们具有最高的优先级别。例如,电源掉电、主存校验错等。

4. 中断升级

中断屏蔽字的另一个作用是可以改变中断优先级,将原级别较低的中断源变成较高的级别,称为中断升级。这实际上是一种动态改变优先级的方法。

这里所说的改变优先次序,是指改变中断的处理次序。中断处理次序和中断响应次序是两个不同的概念,中断响应次序是由硬件排队电路决定的,无法改变。但是,中断处理次序是可以由屏蔽码来改变的,故把屏蔽码看成软排队器。中断处理次序可以不同于中断响应次序。

例如,某计算机的中断系统有 4 个中断源,每个中断源对应一个屏蔽码。表 9-2 为程序优先级与屏蔽码的关系,中断响应的优先次序为 1→2→3→4。根据表 9-2 给出的屏蔽码,中断的处理次序和中断的响应次序是一致的。

表 9-2 程序优先级与屏蔽码

程序级别	屏蔽码			
	1 级	2 级	3 级	4 级
第 1 级	1	1	1	1
第 2 级	0	1	1	1
第 3 级	0	0	1	1
第 4 级	0	0	0	1

根据这一次序,可以看到 CPU 运动的轨迹,如图 9-16 所示。当多个中断请求同时出现时,处理次序与响应次序一致;当中断请求先后出现时,允许优先级别高的中断请求打断优先级别低的中断服务程序,实现中断嵌套。

在不改变中断响应次序的条件下,通过改写屏蔽码可以改变中断处理次序。例如,要使中断处理次序改为 1→4→3→2,则只须使中断屏蔽码改为如表 9-3 所示即可。

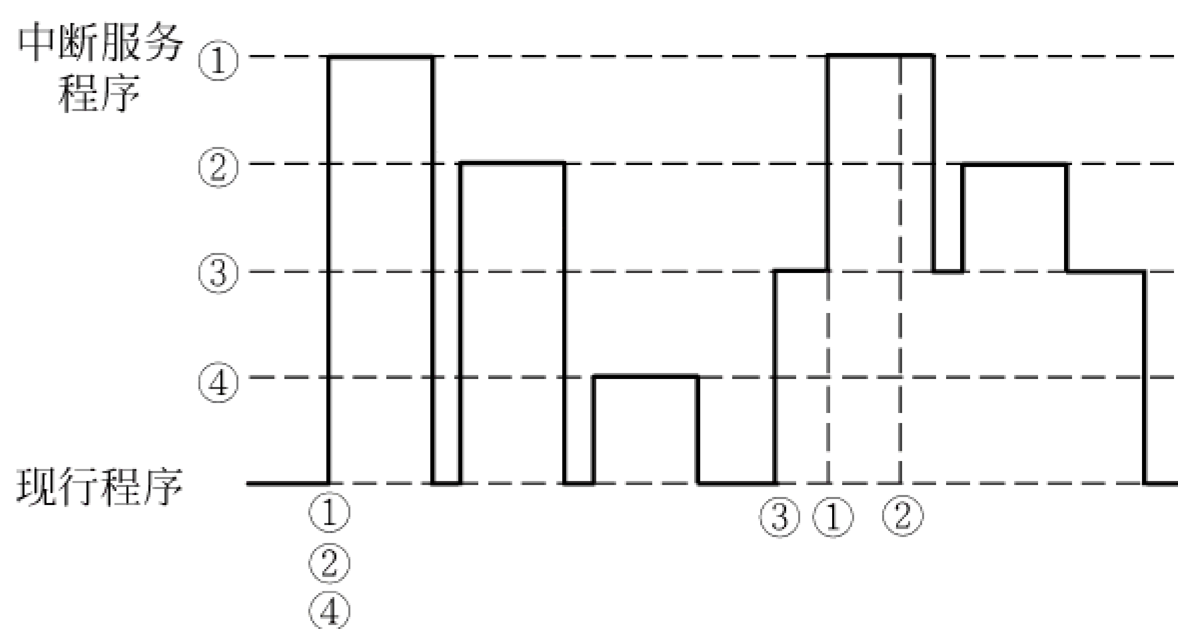


图 9-16 CPU 的运动轨迹

表 9-3 改变处理次序的屏蔽码

程序级别	屏蔽码			
	1 级	2 级	3 级	4 级
第 1 级	1	1	1	1
第 2 级	0	1	0	0
第 3 级	0	1	1	0
第 4 级	0	1	1	1

在同样中断请求的情况下,CPU 的运动轨迹发生了变化,如图 9-17 所示。CPU 正在执行现行程序时,中断源①、②、④同时请求中断服务,显然它们都没有被屏蔽。按照中断优先级别的高低,CPU 首先响应并处理第①级中断请求;当第①级中断处理完后,响应第②级中断请求。CPU 在处理第②级中断时,其屏蔽码对第④级中断是开放的,所以当②级的中断服务程序执行到开中断指令后,立即被④级中断请求打断,CPU 转去执行④级的中断服务程序,待④级的中断服务程序执行完毕后再返回接着执行②级中断服务程序。当第③级中断请求到来并在执行其中断服务程序的过程中,又来了①级中断请求,③级中断服务程序将被①级中断请求打断,转去执行①级中断服务程序。在此过程中,虽然出现了②级中断请求,但因②级的处理级别最低,故不理睬它的请求,直至③级的中断服务程序执行完毕,再响应第②级中断请求。

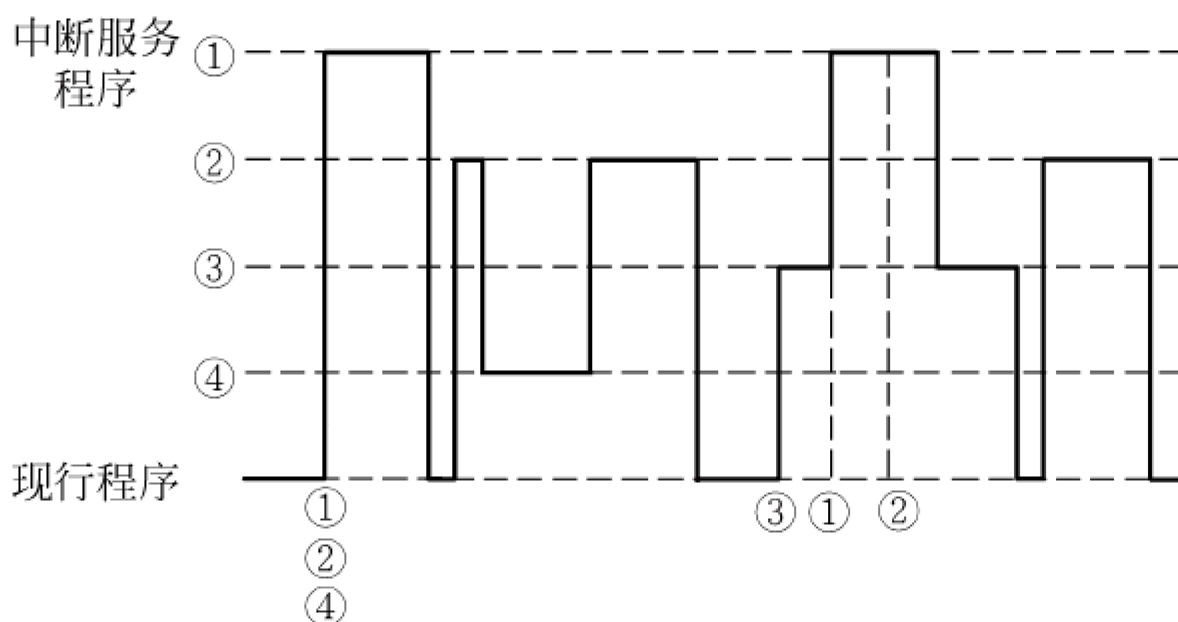


图 9-17 处理次序改变后的 CPU 运动轨迹

由此可见,屏蔽技术向使用者提供了一种手段,即可以用程序控制中断系统,动态地调度多重中断优先处理的次序,从而提高了中断系统的灵活性。

9.3.5 中断全过程

这里所说的中断全过程，指的是从中断源发出中断请求开始，CPU 响应这个请求，现行程序被中断，转至中断服务程序，直到中断服务程序执行完毕，CPU 再返回原来的程序继续执行的整个过程。

大体上可以把中断全过程分为 5 个阶段：中断请求、中断判优、中断响应、中断处理和中断返回。

其中中断处理就是执行中断服务程序，这是中断系统的核心。不同计算机系统的中断处理过程各具特色，但对多数计算机而言，其中断服务程序的流程如图 9-18 所示。图中灰框代表一条指令，白底框代表一段程序，往往不止一条指令。

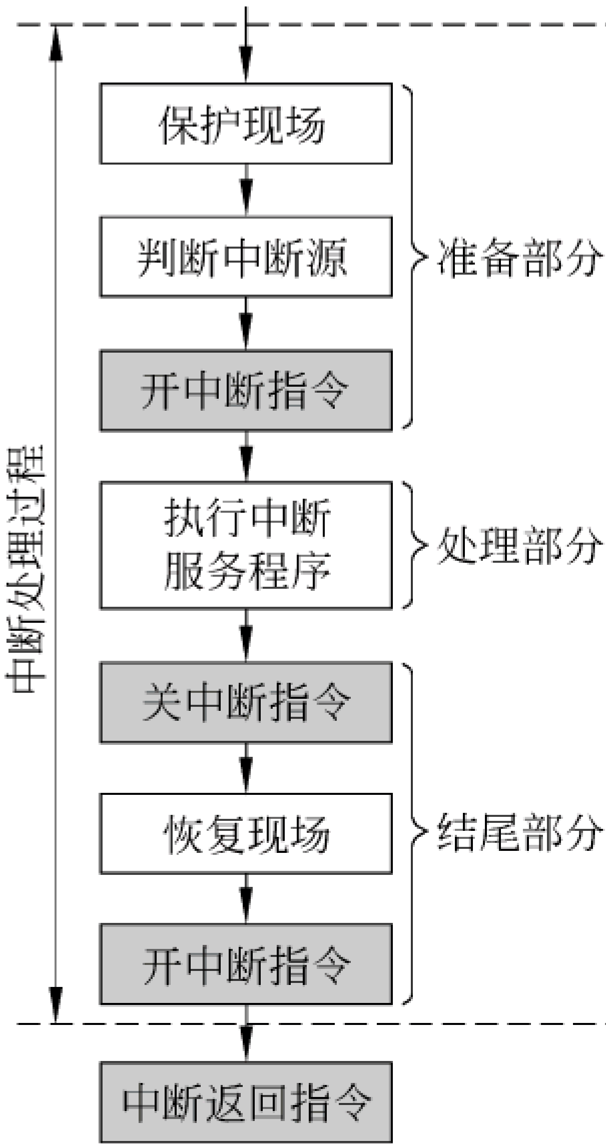


图 9-18 中断服务程序的流程

中断处理过程基本上由 3 个部分组成（以多重中断为例），第一部分为准备部分，其基本功能是保护现场，对于非向量中断方式则需要确定中断源，最后开放中断，允许更高级的中断请求打断低级的中断服务程序；第二部分为处理部分，即真正执行具体的为某个中断源服务的中断服务程序；第三部分为结尾部分，首先要关中断，以防止在恢复现场过程中被新的中断请求打断，接着恢复现场，然后开放中断，以便返回原来的程序后可响应其他的中断请求。中断服务程序的最后一条指令一定是中断返回指令。

注意：保护现场之前的关中断操作由中断隐指令完成。

多重中断与单重中断在中断服务程序的执行中有所不同，表 9-4 列出了两者的区别。

表 9-4 多重中断与单重中断的区别

操作类型	多重中断方式	单重中断方式
中断隐指令	关中断 保存断点及旧 PSW 取中断服务程序入口地址及新 PSW	关中断 保存断点及旧 PSW 取中断服务程序入口地址及新 PSW
中断服务程序	保护现场 送新屏蔽字 开中断	保护现场
	服务处理 （允许响应更高级别请求）	服务处理 （不允许响应更高级别请求）
	关中断 恢复现场及原屏蔽字 开中断 中断返回	恢复现场 开中断 中断返回

9.3.6 程序中断接口结构

具有中断能力的外设接口是由程序查询式接口再加上中断控制机构组成的。简化的中

断式接口如图 9-19 所示。从其逻辑功能来看，这个接口不仅可以保证中断式传送，而且也可以提供程序查询式传送。图中 AB、DB、CB 分别表示地址总线、数据总线和控制总线。

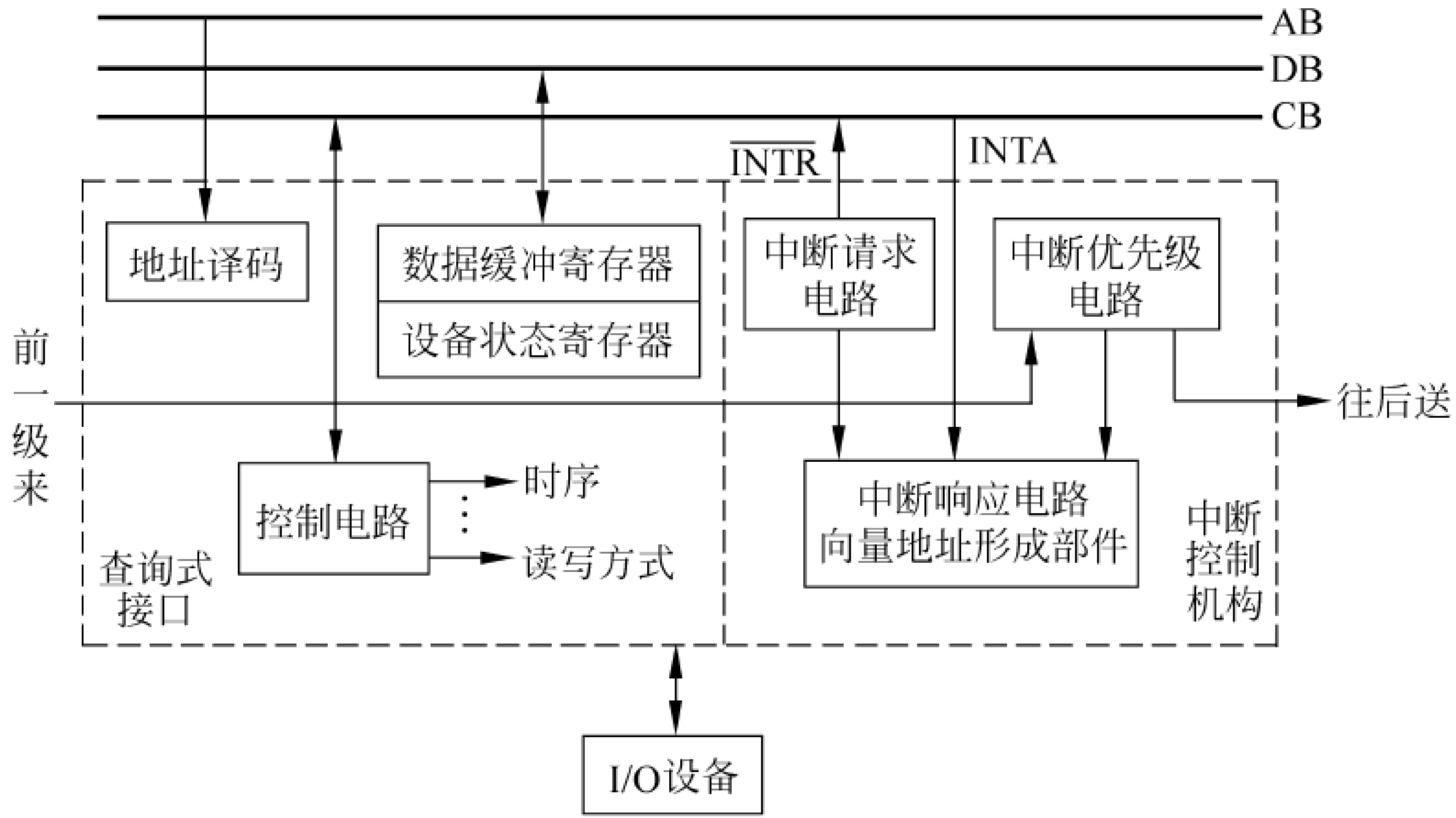


图 9-19　简化的中断式接口

中断控制机构至少应包括有下列几个部分。

(1) 中断请求电路

当中断源有请求且中断开放时，中断请求电路向 CPU 发中断请求信号。

(2) 中断优先级电路

中断优先级电路保证优先级别最高的中断源首先获得 CPU 的服务。

(3) 向量地址形成部件

向量地址形成部件用来产生向量中断时需要的向量地址，并且根据这个向量地址转向该中断源所对应的中断服务程序。

9.4　DMA 方式及其接口

DMA 方式是为了在主存与外设之间实现高速、批量数据交换而设置的。DMA 方式的数据传送直接依靠硬件(DMA 控制器)来实现，不需要执行任何程序。

9.4.1　DMA 方式的基本概念

1. DMA 方式的特点

无论程序查询还是程序中断方式，主要的工作都是由 CPU 执行程序完成的，这需要占用 CPU 时间，因此不能实现高速外设与主机的信息交换。

直接存储器访问(Direct Memory Access，DMA)方式是在外设和主存之间开辟一条“直接数据通道”，在不需要 CPU 干预也不需要软件介入的情况下在两者之间进行的高速数据传送方式。在 DMA 传送方式中，对数据传送过程进行控制的硬件称为 DMA 控制器。当外设需要进行数据传送时，通过 DMA 控制器向 CPU 提出 DMA 传送请求，CPU 响应之后将让出系统总线，由 DMA 控制器接管总线进行数据传送。

DMA 方式具有下列特点：

① 它使主存与 CPU 的固定联系脱钩，主存既可被 CPU 访问，又可被外设访问。

② 在数据块传送时，主存地址的确定、传送数据的计数等都由硬件电路直接实现。

③ 主存中要开辟专用缓冲区，及时供给和接收外设的数据。

④ DMA 传送速度快，CPU 和外设并行工作，提高了系统的效率。

⑤ DMA 在传送开始前要通过程序进行预处理，结束后要通过中断方式进行后处理。

2. DMA 和中断的区别

DMA 与中断的主要区别如下：

① 中断方式是程序切换，需要保护和恢复现场；而 DMA 方式除了开始和结尾时，不占用 CPU 的任何资源。

② 对中断请求的响应时间只能发生在每条指令执行完毕时；而对 DMA 请求的响应时间可以发生在每个机器周期结束时，如图 9-20 所示。

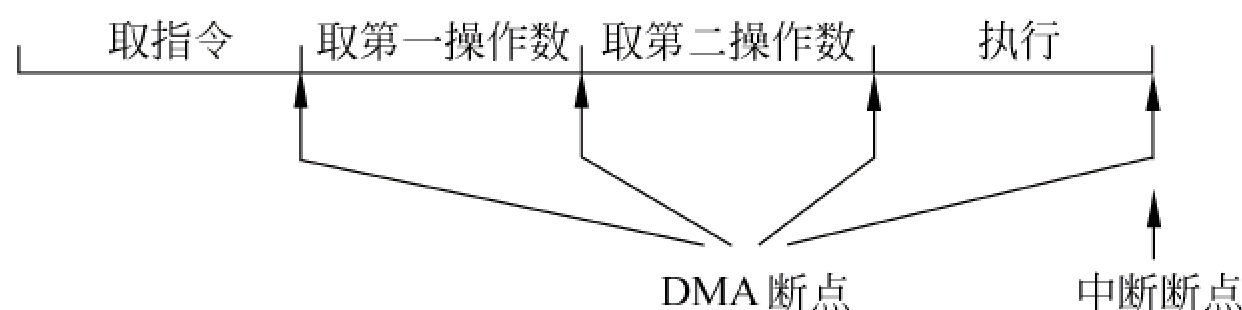


图 9-20　两种请求的响应时刻比较

③ 中断传送过程需要 CPU 的干预；而 DMA 传送过程不需要 CPU 的干预，故数据传输速率非常高，适合于高速外设的成组数据传送。

④ DMA 请求的优先级高于中断请求。

⑤ 中断方式具有对异常事件的处理能力，而 DMA 方式仅局限于完成传送数据块的 I/O 操作。

3. DMA 方式的应用

DMA 方式一般应用于主存与高速外设间的简单数据传送。如磁盘、磁带、光盘等辅助存储器以及其他带有局部存储器的外设、通信设备等都是高速外设。

对磁盘的读写是以数据块为单位进行的，一旦找到数据块起始位置，就将连续地读写。从磁盘中读出数据或往磁盘中写入数据时，一般采用 DMA 方式传送，即直接将数据由主存经数据总线输出到磁盘接口，然后写入盘片；或将数据由盘片读出到磁盘接口，然后经数据总线写入主存。

在大批量数据采集系统中，也可以采用 DMA 方式。

许多计算机系统中选用动态存储器(DRAM)，并用异步方式安排刷新周期。DRAM 的刷新操作可视为存储器内部的数据批量传送，因此，也可采用 DMA 方式实现，将每次刷新请求当成 DMA 请求。CPU 在刷新周期中让出系统总线，按行地址(刷新地址)访问主存，实现各芯片中的一行刷新。利用系统的 DMA 机制实现动态刷新，简化了专门的动态刷新逻辑，提高了主存的利用率。

DMA 传送是直接依靠硬件实现的，可用于快速的数据直传。也正是由于这一点，

DMA 方式本身不能处理较复杂的事件。因此,在某些场合常综合应用 DMA 方式与程序中断方式,二者互为补充。

9.4.2 DMA 接口

DMA 接口相对于查询式接口和中断式接口来说比较复杂,习惯将 DMA 方式的接口电路称为 DMA 控制器。

1. DMA 控制器的功能

在 DMA 传送过程中,DMA 控制器将接管 CPU 的地址总线、数据总线和控制总线,CPU 的主存控制信号被禁止使用。而当 DMA 传送结束后,将恢复 CPU 的一切权力并开始执行其操作。由此可见,DMA 控制器必须具有控制系统总线的能力,也就是说能够像 CPU 一样输出地址信号,接收或发出控制信号,输入或输出数据信号。

DMA 控制器在外设与主存之间直接传送数据期间,完全代替 CPU 进行工作,它的主要功能有:

① 接受外设发出的 DMA 请求,并向 CPU 发出总线请求。

② 当 CPU 响应此总线请求,发出总线响应信号后,接管对总线的控制,进入 DMA 操作周期。

③ 确定传送数据的主存单元地址及传送长度,并能自动修改主存地址计数值和传送长度计数值。

④ 规定数据在主存与外设之间的传送方向,发出读写或其他控制信号,并执行数据传送的操作。

⑤ 向 CPU 报告 DMA 操作的结束。

2. DMA 控制器的基本组成

图 9-21 给出了一个简单的 DMA 控制器框图,它由以下几部分组成。

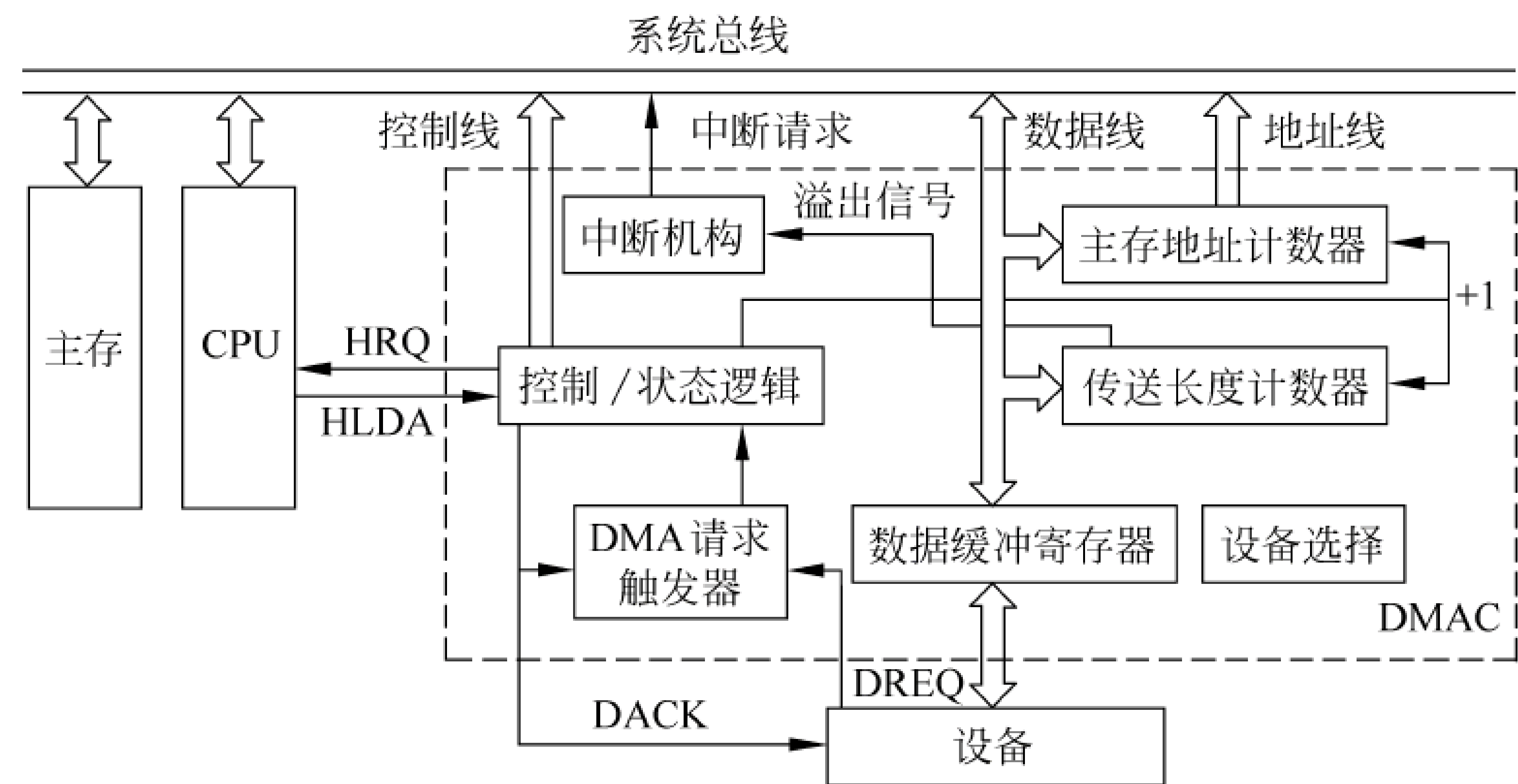


图 9-21 简单的 DMA 控制器

(1) 主存地址计数器

主存地址计数器用来存放待交换数据的主存地址。该计数器的初始值为主存缓冲区的

首地址，当 DMA 传送时，每传送一个数据，将地址计数器加 1，从而以增量方式给出主存中要交换的一批数据的地址，直至这批数据传送完毕为止。

(2) 传送长度计数器

传送长度计数器用来记录传送数据块的长度。其初始值为传送数据的总字数或总字节数，每传送一个字或一个字节，计数器自动减 1，当其内容为 0 时表示数据已全部传送完毕。也有些 DMA 控制器中，初始时将字数或字节数求补之后送计数器，每传送一个字或一个字节，计数器加 1，当计数器溢出时，表示数据传送完毕。

(3) 数据缓冲寄存器

数据缓冲寄存器用来暂存每次传送的数据。输入时，数据由外设（如磁盘）先送往数据缓冲寄存器，再通过数据总线送到主存。反之，输出时，数据由主存通过数据总线送到数据缓冲寄存器，然后再送到外设。

(4) DMA 请求触发器

DMA 请求触发器的作用是每当外设准备好数据后给出一个控制信号，使 DMA 请求触发器置位。

(5) 控制/状态逻辑

它由控制和时序电路以及状态标志等组成，用于指定传送方向，修改传送参数，并对 DMA 请求信号和 CPU 响应信号进行协调和同步。

(6) 中断机构

当一个数据块传送完毕后触发中断机构，向 CPU 提出中断请求，CPU 将进行 DMA 传送的结尾处理。

有些商品化的 DMA 控制器芯片中看似并没有设置中断机构，但并不代表 DMA 的结尾处理不需要中断的参与，因为系统一定还同时配有中断控制器芯片，两个芯片共同完成了 DMA 的功能。

3. DMA 控制器的引出线

DMA 控制器必须有下列引出线。

(1) 地址线

在 DMA 方式下，地址线呈输出状态，可对主存进行地址选择；在 CPU 方式下，地址线呈输入状态，可对 DMA 控制器中的有关寄存器进行寻址。

(2) 数据线

在 DMA 方式下，用它进行数据传送；在 CPU 方式下，可对 DMA 控制器的有关寄存器进行编程。

(3) 控制数据传送方式的信号线

存储器读信号$\overline{MEMR}$、存储器写信号$\overline{MEMW}$、外设读信号$\overline{IOR}$和外设写信号$\overline{IOW}$。

当数据从外设写入主存时，$\overline{MEMW}$和$\overline{IOR}$同时有效；而当数据从主存读出送外设时，$\overline{MEMR}$和$\overline{IOW}$将同时有效。

(4) DMA 控制器与外设之间的联络信号线

DMA 请求信号 DREQ（输入）是外设向 DMA 控制器提出 DMA 操作的申请信号。

DMA 响应信号 DACK（输出）是 DMA 控制器给提出 DMA 请求的外设的应答信号。

(5) DMA 控制器与 CPU 之间的联络信号线

总线请求信号 HRQ(输出)是 DMA 控制器向 CPU 请求使用总线的信号。

总线响应信号 HLDA(输入)是 CPU 向 DMA 控制器表示响应总线请求的信号。

4. DMA 控制器的连接和传送

图 9-22 给出了 DMA 控制器与 CPU 及主存、外设之间的连接框图。在进行 DMA 操作之前应先对 DMA 控制器编程。例如,确定传送数据的主存起始地址、要传送的字节数以及传送方式,是由外设将数据写入主存还是从主存将数据读出送外设。下面以外设将一个数据块写入主存的操作为例,简述 DMA 控制器的操作过程。

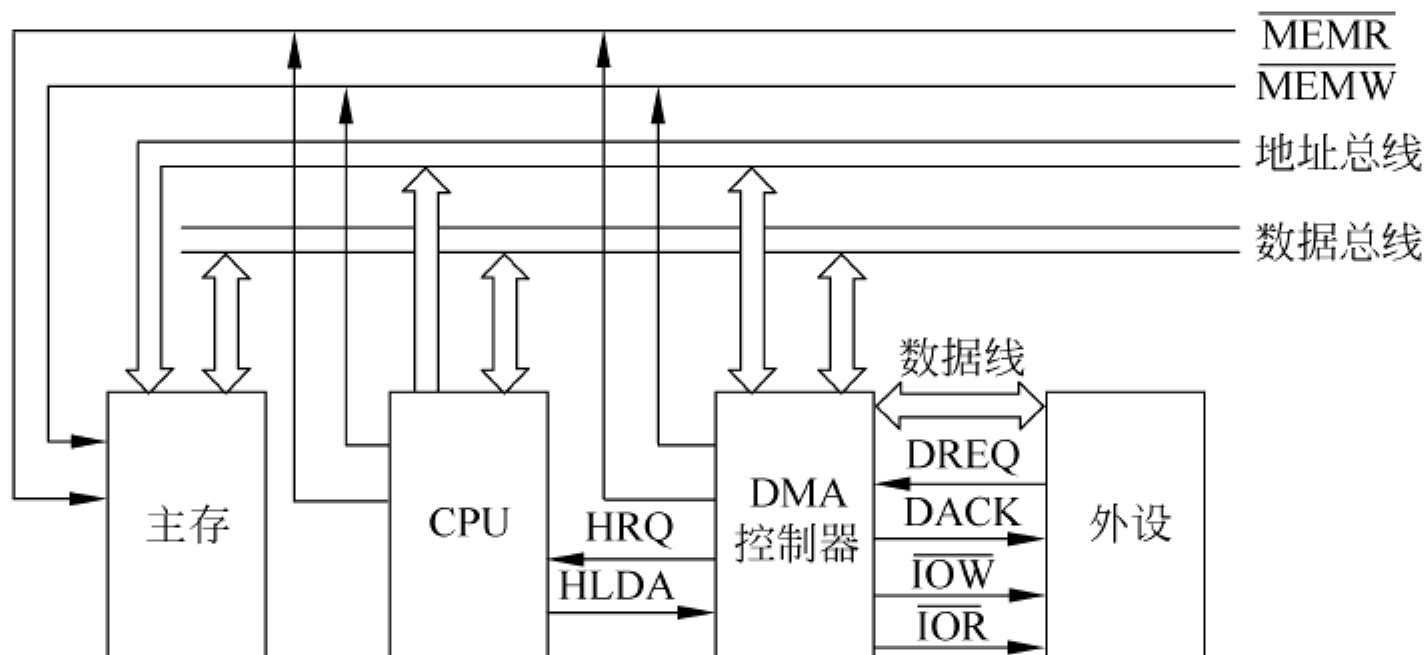


图 9-22　DMA 控制器与 CPU 及主存、外设之间的连接

① 由外设向 DMA 控制器发出 DMA 请求信号 DREQ。

② DMA 控制器向 CPU 发出总线请求信号 HRQ。

③ CPU 向 DMA 控制器发出总线响应信号 HLDA,此时 DMA 控制器获取了总线的控制权。

④ DMA 控制器向外设发出 DMA 响应信号 DACK,表示 DMA 控制器已控制了总线,允许外设与主存交换数据。

⑤ DMA 控制器按主存地址计数器的内容发出地址信号作为主存地址的选择,同时主存地址计数器的内容加 1。

⑥ DMA 控制器发出 $\overline{IOR}$ 信号到外设,将外设数据读入数据缓冲寄存器,同时发出 $\overline{MEMW}$ 信号,将数据缓冲寄存器中的数据写入选中的主存单元。

⑦ 传送长度计数器减 1。

重复⑤～⑦步骤,直到字节计数器减到 0 为止,数据块的 DMA 方式传送工作宣告完成。这时,DMA 控制器的 HRQ 降为低电平,总线控制权交还 CPU。

9.4.3　DMA 传送方法与传送过程

1. DMA 传送方法

DMA 控制器与 CPU 通常采用以下 3 种方法使用主存。

(1) CPU 停止访问主存法

这是最简单的 DMA 方法。这种方法是用 DMA 请求信号迫使 CPU 让出总线控制权。

CPU 在现行机器周期执行完成之后，使其数据、地址总线处于三态，并输出总线批准信号。每次 DMA 请求获得批准，DMA 控制器获得总线控制权以后，连续占用若干个存取周期（总线周期）进行成组连续的数据传送，直至批量传送结束，DMA 控制器才把总线控制权交回 CPU。在 DMA 操作期间，CPU 处于保持状态，停止访问主存，仅能进行一些与总线无关的内部操作。图 9-23(a)是这种传送方法的时间图，该方法只适用于高速外设的成组传送。

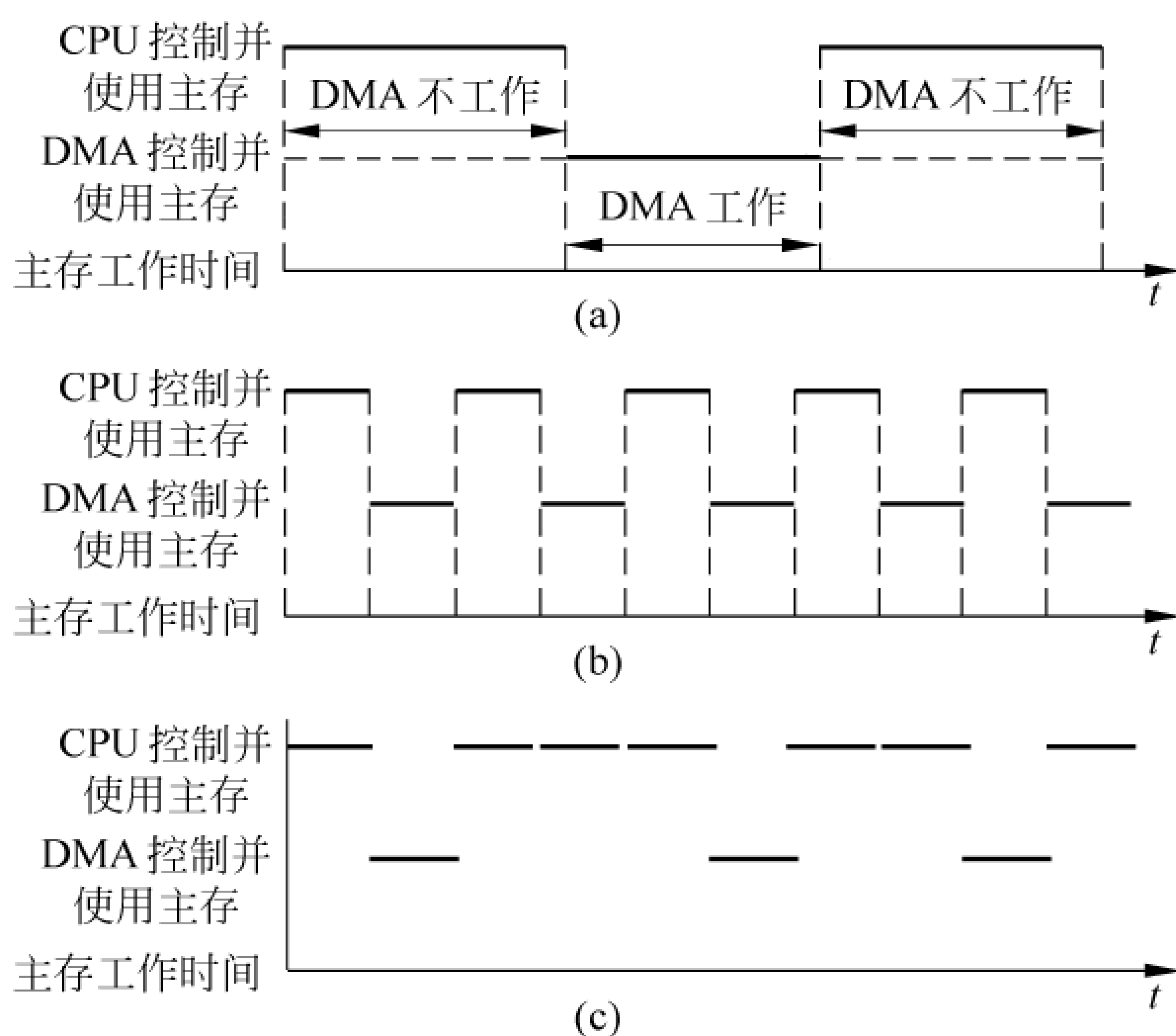


图 9-23　DMA 传送方法

当外设的数据传输率接近于主存工作速度时，或者 CPU 除了等待 DMA 传送结束并无其他事可干（如单用户状态下的个人计算机）时，常采用这种方法。它可以减少系统总线控制权的交换次数，有利于提高输入输出的速度。

(2) 存储器分时法

把原来的一个存取周期分成两个时间片，一片分给 CPU，一片分给 DMA，使 CPU 和 DMA 交替地访问主存。这种方法无须申请和归还总线，使总线控制权的转移几乎不需要什么时间，所以对 DMA 传送来讲效率是很高的，而且 CPU 既不停止现行程序的运行，也不进入保持状态，在 CPU 不知不觉中便进行了 DMA 传送；但这种方法需要主存在原来的存取周期内为两个部件服务，如果要维持 CPU 的访存速度不变，就要求主存的工作速度提高一倍。另外，由于大多数外设的速度都不能与 CPU 相匹配，所以供 DMA 使用的时间片可能成为空操作，将会造成一些不必要的浪费。图 9-23(b)是这种方法的时间图。

(3) 周期挪用法

周期挪用法是前两种方法的折中。当外设没有 DMA 请求时，CPU 按程序要求访问主存；一旦外设有 DMA 请求并获得 CPU 批准后，CPU 让出一个周期的总线控制权，由 DMA 控制器控制系统总线，挪用一个存取周期进行一次数据传送，传送一个字节或一个字；然后，DMA 控制器将总线控制权交回 CPU，CPU 继续进行自己的操作，等待下一个 DMA 请求的到来。重复上述过程，直至数据块传送完毕。如果在同一时刻，发生 CPU 与 DMA 的访存冲突，那么优先保证 DMA 工作，而 CPU 等待一个存取周期，如图 9-23(c)所示。若 DMA 传送期间 CPU 无须访存，则周期挪用对 CPU 执行程序无任何影响。

当主存工作速度高出外设较多时,采用周期挪用法可以提高主存的利用率,对CPU的影响较小,因此,高速主机系统常采用这种方法。根据主存的存取周期与磁盘的数据传输率,可以计算出主存操作时间的分配情况:有多少时间需用于DMA传送(被挪用),有多少时间可用于CPU访存。这在一定程度上反映了系统的处理效率。

2. DMA传送过程

DMA的传送过程可分为3个阶段:DMA传送前的预处理、数据传送和传送后的结束处理。

(1) DMA预处理

在DMA传送之前必须要做准备工作,即初始化。这是由CPU来完成的。CPU首先执行几条I/O指令,用于测试外设的状态、向DMA控制器的有关寄存器置初值、设置传送方向、启动该外部设备等。

在这些工作完成之后,CPU继续执行原来的程序,在外设准备好发送的数据(输入)或接收的数据已处理完毕(输出)时,外设向DMA控制器发DMA请求,再由DMA控制器向CPU发总线请求。

(2) 数据传送

DMA的数据传送可以以单字节(或字)为基本单位,也可以以数据块为基本单位。对于以数据块为单位的传送,DMA控制器占用总线后的数据输入和输出操作都是通过循环来实现的,其传送过程如图9-24所示。

注意:*图9-24所示的流程图不是由CPU执行程序实现的,而是由DMA控制器实现的。*

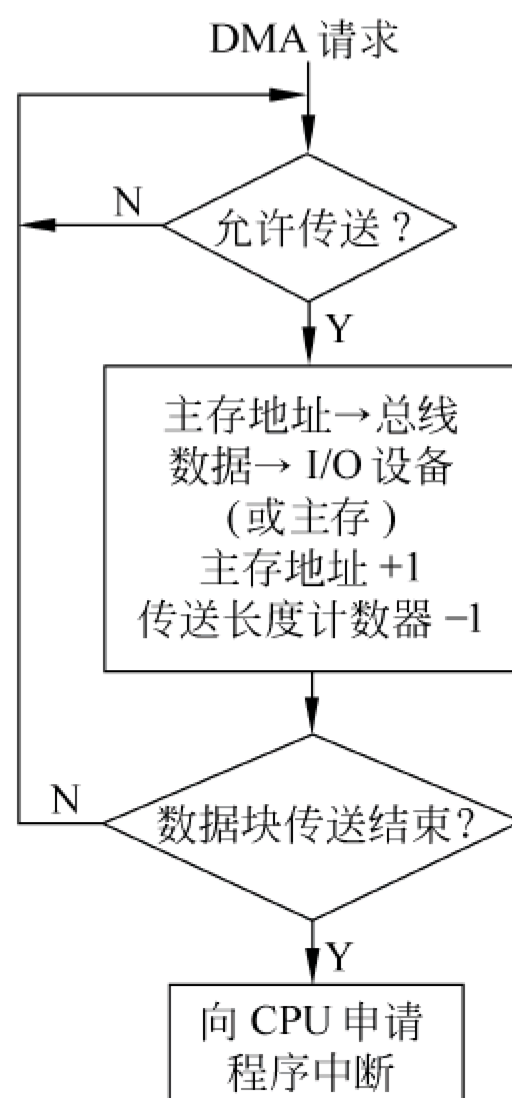


图9-24 DMA的数据传送过程

(3) DMA后处理

当传送长度计数器计到0时,DMA操作结束,DMA控制器向CPU发中断请求,CPU停止原来程序的执行,转去执行中断服务程序做DMA结束处理工作。

9.5 通道控制方式

在大型计算机系统中,所连接的I/O设备数量多,输入输出频繁,要求整体的速度快,单纯依靠主CPU采取程序中断和DMA等控制方式已不能满足要求,于是通道控制方式被引入计算机系统。

9.5.1 通道的基本概念

1. 通道控制方式与DMA方式的区别

通道控制方式是DMA方式的进一步发展,实质上,通道也是实现外设和主存之间直接交换数据的控制器。与DMA控制器相比,两者的主要区别在于:

① DMA 控制器是通过专门设计的硬件控制逻辑来实现对数据传送的控制；而通道则是一个具有特殊功能的处理器，它具有自己的指令和程序，通过执行通道程序来实现对数据传送的控制，故通道具有更强的独立处理数据输入输出的功能。

② DMA 控制器通常只能控制一台或少数几台同类设备；而一个通道则可以同时控制许多台同类或不同类的设备。

2. 通道的功能

在第 1 章已经引出了典型的具有通道的计算机结构。从图 1-4 中可以看出，主机可以接若干个通道，一个通道可以接若干个设备控制器，一个设备控制器又可以接一台或多台外部设备。因此，从逻辑结构上讲，通道控制方式具有 4 级连接：主机→通道→设备控制器→外部设备。

通道是一种高级的 I/O 控制部件，它在一定的硬件基础上利用软件手段实现对 I/O 的控制和传送，更多地免去了 CPU 的介入，从而使主机和外设的并行工作程度更高。当然，通道并不能完全脱离 CPU，它还要受到 CPU 的管理，如启动、停止等，而且通道还应该向 CPU 报告自己的状态，以便 CPU 决定下一步的处理。

通道应具有以下几个方面的功能：

① 接受 CPU 的 I/O 指令，按指令要求与指定的外设进行联系。

② 从主存取出属于该通道程序的通道指令，经译码后向设备控制器和设备发送各种命令。

③ 实施主存和外设间的数据传送，如为主存或外设装配和拆卸信息，提供数据中间缓存的空间以及指示数据存放的主存地址和传送的数据量。

④ 从外设获得设备的状态信息，形成并保存通道本身的状态信息，根据要求将这些状态信息送到主存的指定单元，供 CPU 使用。

⑤ 将外设的中断请求和通道本身的中断请求按次序及时报告 CPU。

9.5.2 通道的类型与结构

1. 通道类型

按照通道独立于主机的程度，可分为结合型通道和独立型通道两种类型。结合型通道在硬件结构上与 CPU 结合在一起，借助于 CPU 的某些部件作为通道部件来实现外设与主机的信息交换。这种通道结构简单，成本较低，但功能较弱。独立型通道完全独立于主机对外设进行管理和控制。这种通道功能强，但设备成本高。

按照输入输出信息的传送方式，通道可分为字节多路通道、选择通道和数组多路通道 3 种类型。

(1) 字节多路通道

字节多路通道是一种简单的共享通道，用于连接与管理多台低速设备，以字节交叉方式传送信息，其传送方式如图 9-25 所示。字节多路通道先选择设备 A，为其传送一个字节 A_1；然后选择设备 B，传送字节 B_1；再选择设备 C，传送字节 C_1。再交叉地传送 A_2，B_2，C_2，…所以字节多路通道的功能好比一个多路开关，交叉(轮流)地接通各台设备。

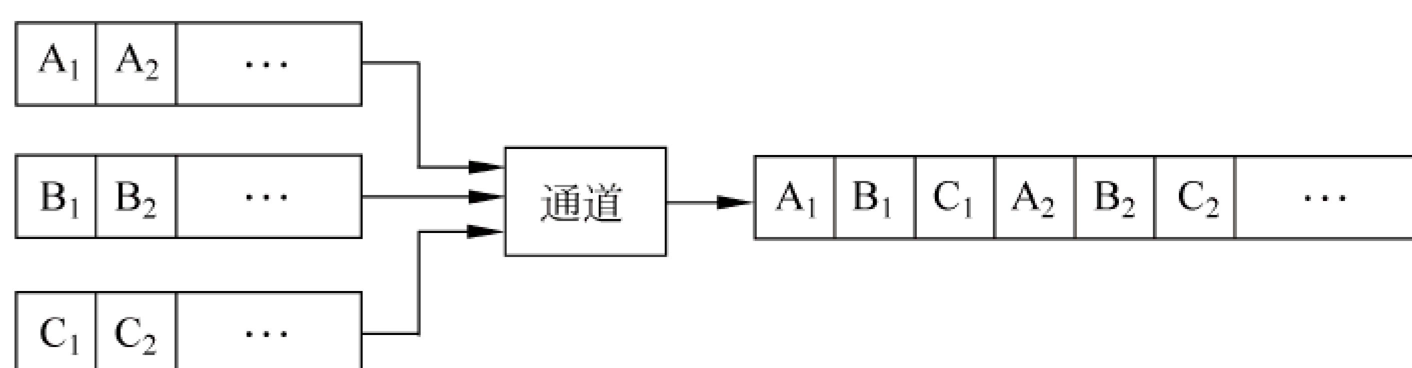


图 9-25　字节多路通道传送方式

一个字节多路通道,包括多个按字节方式传送信息的子通道。每个子通道服务于一个设备控制器,每个子通道都可以独立地执行通道程序。各个子通道可以并行工作,但是,所有子通道的控制部分是公共的,各个子通道可以分时地使用。

通道不间断地、轮流地启动每个设备控制器,当通道为一个设备传送完一个字节后,就转去为另一个设备服务。当通道为某一设备传送时,其他设备可以并行地工作,准备需要传送的数据字节或处理收到的数据字节。这种轮流服务是建立在主机的速度比外设的速度高得多的基础之上的,它可以提高系统的工作效率。

(2) 选择通道

对于高速设备,字节多路通道显然是不合适的。选择通道又称高速通道,在物理上它也可以连接多个设备,但这些设备不能同时工作,在一段时间内通道只能选择一台设备进行数据传送,此时该设备可以独占整个通道。因此,选择通道一次只能执行一个通道程序,只有当它与主存交换完信息后,才能再选择另一台外部设备并执行该设备的通道程序。如图 9-26 所示,选择通道先选择设备 A,成组连续地传送 $A_1A_2\cdots$,当设备 A 传送完毕后,选择通道又选择通道 B,成组连续地传送 $B_1B_2\cdots$,再选择设备 C,成组连续地传送 $C_1C_2\cdots$。

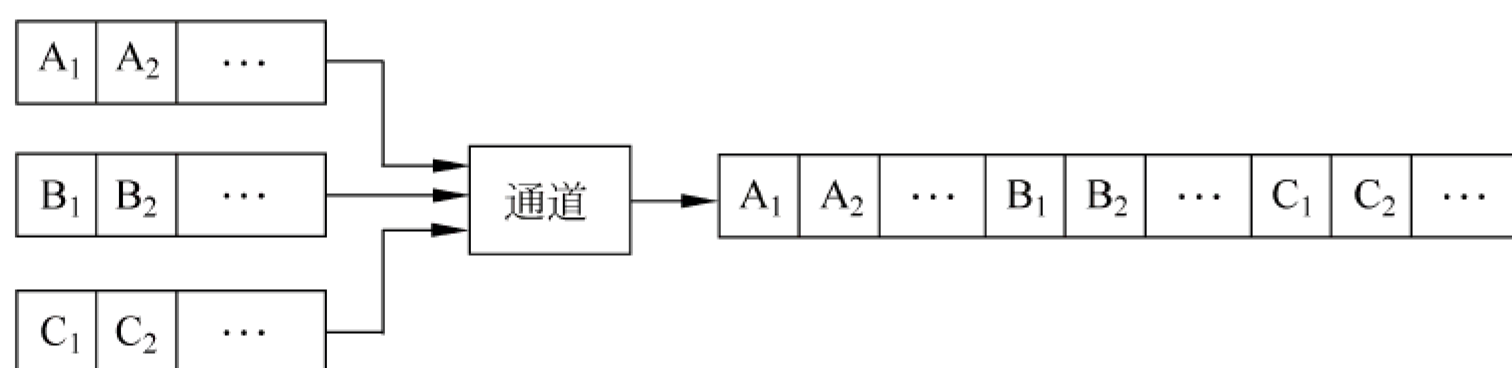


图 9-26　选择通道传送方式

选择通道主要用于连接高速外设,如磁盘、磁带等,信息以成组方式高速传送。但是,在数据传送过程中还有一些辅助操作,如磁盘机的寻道等,此时会使通道处于等待状态,所以虽然选择通道具有很高的数据传输速率,但整个通道的利用率并不高。

(3) 数组多路通道

数组多路通道是把字节多路通道和选择通道的特点结合起来的一种通道结构。它的基本思想是:当某设备进行数据传送时,通道只为该设备服务;当设备在执行辅助操作时,通道暂时断开与这个设备的连接,挂起该设备的通道程序,去为其他设备服务。

数组多路通道有多个子通道,既可以执行多路通道程序,即像字节多路通道那样,所有子通道分时共享总通道,又可以用选择通道那样的方式成组地传送数据;既具有多路并行操作的能力,又具有很高的数据传输速率,使通道的效率充分得到发挥。

选择通道和数组多路通道都适用于连接高速外设，但前者的数据宽度是不定长的数据块，后者的数据宽度是定长的数据块。3种类型通道的比较见表9-5。3种类型的通道组织在一起，可配置若干台不同种类、不同速度的I/O设备，使计算机的I/O组织更合理、功能更完善、管理更方便。

表9-5　3种类型通道的比较

性能＼通道类型	字节多路	选择	数组多路
数据宽度	单字节	不定长块	定长块
适用范围	大量低速设备	优先级高的高速设备	大量高速设备
工作方式	字节交叉	独占通道	成组交叉
共享性	分时共享	独占	分时共享
选择设备次数	多次	一次	多次

通道在单位时间内传送的位数或字节数称为通道的数据传输率或流量，它标志了计算机系统中的系统吞吐量，也表明了通道对外设的控制能力和效率。在单位时间内允许传送的最大字节数或位数称为通道的最大数据传输率或通道极限流量，它是设计通道的最大依据。

字节多路通道的实际流量是该通道上所有设备的数据传输率之和。而选择通道和数组多路通道在一段时间内只能为一台设备传送数据，这时的通道流量就等于这台设备的数据传输率。因此，这两种通道的实际流量等于连接在这个通道上的所有设备中流量最大的那一个。

2. 通道的结构

通道的一般逻辑结构如图9-27所示，其中CSWR、CAWR、CCWR是3个重要的寄存器。CCWR是通道命令字寄存器，它用来存放通道命令字(CCW)。CCW是控制I/O操作的关键参数，一条条的通道命令字(通道指令)构成通道程序，放在主存中。CAWR是通道地址字寄存器，它指出了通道程序在主存中的起始地址，工作时通道就依照这个地址到主存中取出CCW并加以执行。CSWR是通道状态字寄存器，记录了通道程序执行后本通道和相应设备的各种状态信息，这些信息称为通道状态字(CSW)。CSW通常放在主存的固定单元中，此专用单元的内容在执行下一个I/O指令或中断之前是有效的，可供CPU了解通道、设备状态和操作结束的原因。

9.5.3　通道程序

1. 通道指令

通道指令也就是通道命令字(CCW)，它用来编制通道程序，并由管理程序将它存放在主存的任何地方。为了使通道能够快速地找到通道程序，用通道地址字(CAW)指出通道程序的起始地址。在主CPU执行“启动I/O”指令启动指定通道后，通道将执行通道程序来实现具体的I/O操作，直到组成通道程序的全部CCW执行完毕时，这次I/O传送就算完成了。通道指令格式简单，功能专一，一般带有很强的面向外部设备的特征。

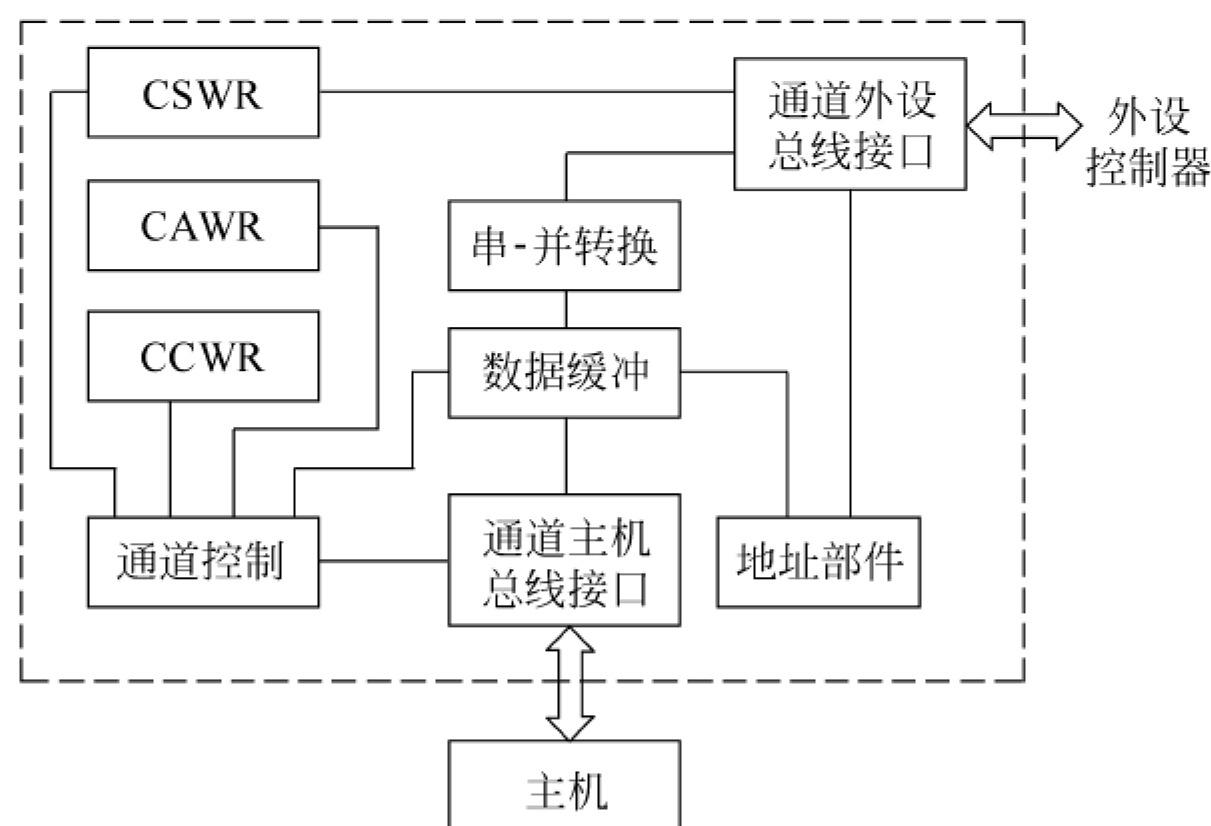


图 9-27 通道逻辑结构

通道指令的功能和格式因计算机不同而异，下面介绍 IBM 4300 的通道指令格式，其格式如图 9-28 所示。

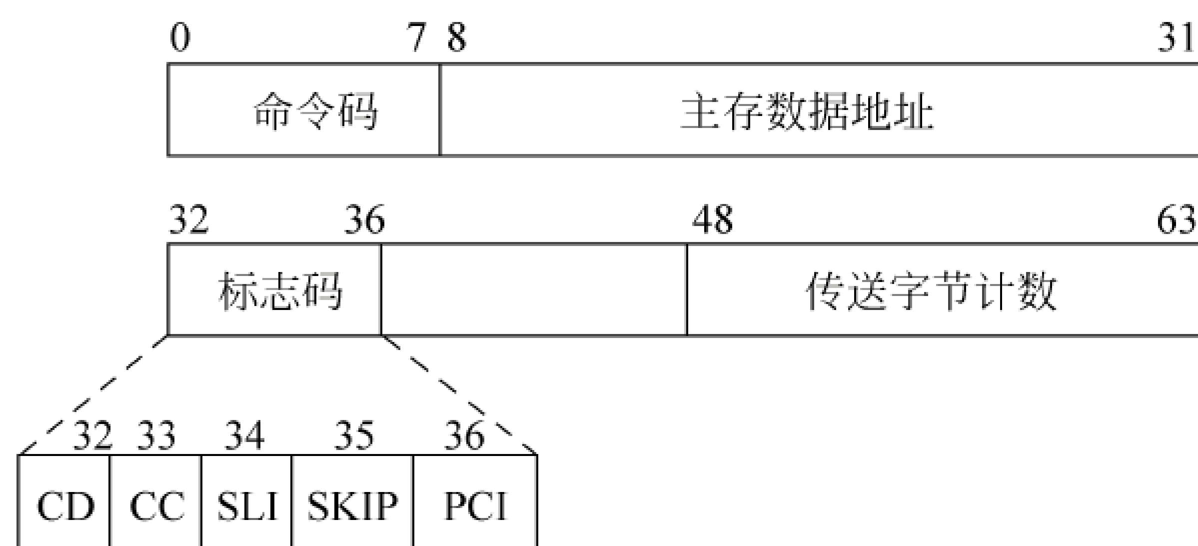


图 9-28 IBM 4300 通道指令格式

这是一个双字长(64 位)的指令，共分 5 个字段。

(1) 命令码

命令码字段相当于机器指令的操作码，由它决定通道和设备执行什么操作。

(2) 主存数据地址

通道指令中的 8～31 位(共 24 位)给出本次 I/O 传送操作(读、写、反读)时主存缓冲区的首地址，在数据传送过程中，每传送一个字(或字节)，数据地址修改一次(加 1 或减 1)。

(3) 传送字节计数

通道指令中的 48～63 位(共 16 位)，用来表示通道执行 I/O 操作时所传送的数据块长度，通常以字节为单位。其值可以是传送的字节数，每传送一次计数值减 1；也可以是传送字节数的补数，每传送一次计数值加 1，当计数值为全 0 时，表示数据块传送完毕。

(4) 标志码

通道指令中的 32～36 位(共 5 位)，用来定义通道程序的链接方式或通道命令的操作特征，统称为特征位，各位的含义如下。

① 数据链特征：用 CD 表示。CD=1，表示接下去的一条通道指令也是数据传送命令。执行完本条通道指令后不必断开与通道的逻辑联系，接着取出下一条通道指令来执行即可。

第二条通道指令的命令码和第一条的命令码相同。

② 命令链特征：用 CC 表示。CC＝1，表示本条通道指令执行完毕，接着有不同操作命令的通道指令要执行。执行完本条通道指令后要断开与通道的逻辑联系，接着取下一条通道指令。前后两条通道指令的命令码是不相同的。

由此可见，只要通道指令中的 CD 或 CC 位为 1，就表示通道程序还没有结束；当 CD 和 CC 位全为 0 时，表示本条通道指令是通道程序的最后一条指令，通道程序将结束。

③ 封锁错误长度特征：用 SLI 表示。所谓长度错，是指当通道指令中所给定的传送字节个数与外部设备请求传送的字节个数不相等时，通道指令执行完毕将产生长度错误标志，并向 CPU 发中断请求。若 SLI＝1，即使产生了长度错误标志，也不发送错误信号，不产生中断请求，继续执行通道指令。

④ 封锁写入主存特征：用 SKIP 表示。SKIP＝1 时，禁止将外部设备读出的数据写入主存。本特征位若与数据链特征位连用，则可从外部设备的一批连续数据中任选一部分写入主存。

⑤ 程序控制中断特征：用 PCI 表示。PCI＝1，表示执行本通道指令时允许产生一个中断条件。

2. 通道程序举例

通道程序由一条或几条 CCW 组成，在进行通道程序设计时，要特别注意命令码和标志码的应用。例如，在对磁盘机进行读写操作前，要使用控制命令查找磁盘地址，这个地址（含柱面号、盘面号、扇区号等）被包含在控制命令的数据地址字段中。根据此命令使磁盘机进行寻址工作，当找到指定的磁盘数据区时，通道才开始执行真正的对磁盘机进行读写操作的通道程序。下面通过两个例子看通道程序。

例 9-1　磁盘写入操作。

把主存中 3 个长度分别为 128 个字节、96 个字节和 256 个字节的数据块写入到磁盘机的指定地址中。这 3 个数据块的主存起始地址分别为 002000H、002100H、002200H。同时，假设磁盘数据区地址已通过前面的通道程序找到了，因此完成磁盘写入操作的通道程序如表 9-6 所示。

表 9-6　磁盘机写入操作通道程序举例

通道指令	命　令	主存地址（十六进制）	标志码（二进制）	字节计数（十进制）
CCW_1	磁盘写	002000	10000	128
CCW_2	无用	002100	10000	96
CCW_3	无用	002200	00000	256

例 9-2　磁带读出操作。

从磁带机读出一个数据块，总长度为 256 个字节，分别放到主存的两个地方，其中第一个位置的起始地址为 005000H，存放数据块开始的 120 个字节；第二个位置的起始地址为 006000H，存放数据块的最后 80 个字节，数据块中间部分的 56 个字节不送入主存中，其通道程序如表 9-7 所示。

表 9-7　磁带机读出操作通道程序举例

通道指令	命　令	主存地址(十六进制)	标志码(二进制)	字节计数(十进制)
CCW_1	磁带读	005000	10000	120
CCW_2	无用	无用	10010	56
CCW_3	无用	006000	00000	80

9.5.4　通道工作过程

通道完成一次数据传输的主要过程分为如下3步:

① 在用户程序中使用访管指令进入管理程序,由CPU通过管理程序组织一个通道程序,并启动通道。

② 通道执行CPU为它组织的通道程序,完成指定的数据输入输出工作。

③ 通道程序结束后向CPU发中断请求。CPU响应这个中断请求后,第二次调用管理程序对中断请求进行处理。

这样,每完成一次输入输出工作,CPU只需要两次调用管理程序,大大减少了对用户程序的打扰。CPU执行用户程序和管理程序,通道执行通道程序的时间关系如图9-29所示。

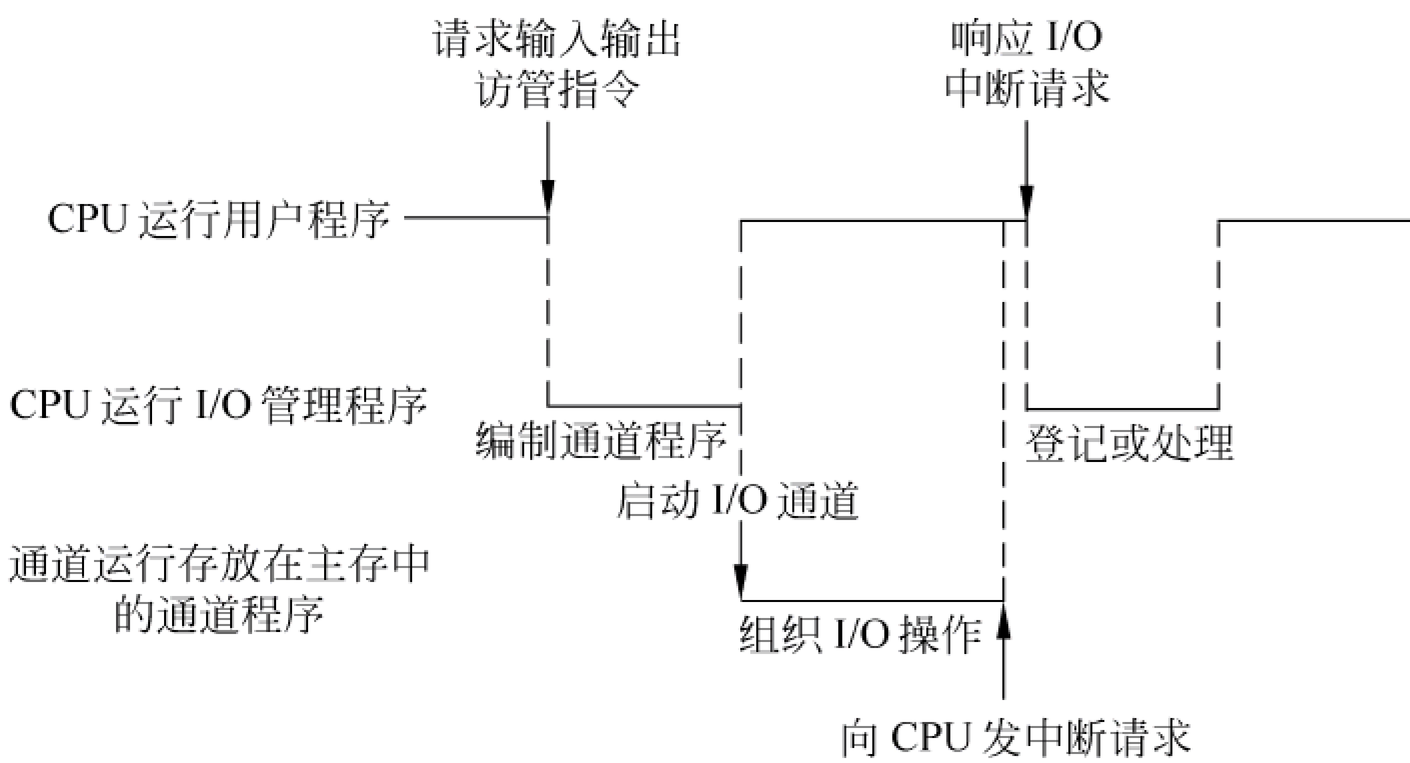


图9-29　用户程序、管理程序和通道程序的执行时间关系

习　　题

9-1　什么是计算机的输入输出系统?输入输出设备有哪些编址方式?有什么特点?

9-2　什么是I/O接口?I/O接口有哪些特点和功能?接口有哪些类型?

9-3　并行接口和串行接口实质上的区别是什么?其界面如何划分?各有什么特点?

9-4　程序查询方式、程序中断方式、DMA方式各自适用什么范围?下面这些结论正确吗?为什么?

(1) 程序中断方式能提高CPU利用率,所以在设置了中断方式后就没有再应用程序查询方式的必要了。

(2) DMA方式能处理高速外部设备与主存间的数据传送,高速工作性能往往能覆盖低速工作要求,所以DMA方式可以完全取代程序中断方式。

9-5　什么是程序查询I/O传送方式?试举例说明其工作原理,它有哪些优缺点?

9-6　图9-5是以程序查询方式实现与多台设备进行数据交换的流程图,试分析这种处理方式存在的

问题以及改进措施。

9-7　如果采用程序查询方式从磁盘上输入一组数据，设主机执行指令的平均速度为100万条指令每秒，试问从磁盘上读出相邻两个数据的最短允许时间间隔是多少？若改为中断式输入，这个间隔是更短些还是更长些？由此可得出什么结论？

9-8　在程序查询方式的输入输出系统中，假设不考虑处理时间，每一个查询操作需要100个时钟周期，CPU的时钟频率为50MHz。现有鼠标和硬盘两个设备，而且CPU必须每秒对鼠标进行30次查询，硬盘以32位字长为单位传输数据，即每32位被CPU查询一次，传输率为2MB/s。求CPU对这两个设备查询所花费的时间比率，由此可得出什么结论？

9-9　什么是中断？外部设备如何才能产生中断？

9-10　中断为什么要判优？有哪些具体的判优方法？各有什么优缺点？

9-11　CPU响应中断应具备哪些条件？

9-12　什么叫中断隐指令？中断隐指令有哪些功能？中断隐指令如何实现？

9-13　什么是中断向量？中断向量如何形成？向量中断和非向量中断有何差异？

9-14　在程序中断处理中，要做到现行程序向中断服务程序过渡和中断服务程序执行完毕返回现行程序，必须进行哪些关键性操作？一般采用什么方法实现这些操作？

9-15　假定某计算机的中断处理方式是将断点存入00000Q单元，并从77777Q单元取出指令(即中断服务程序的第一条指令)执行。试排出完成此功能的中断周期微操作序列，并判断出中断服务程序的第一条指令是何指令(假定主存容量为2^{15}个单元)？

9-16　假设有设备1和设备2两个设备，其优先级为设备1＞设备2，若它们同时提出中断请求，试说明中断处理过程，画出其中断处理过程示意图，并标出断点。

9-17　设某计算机有4个中断源，优先顺序按1→2→3→4降序排列，若1、2、3、4中断源的服务程序中对应的屏蔽字分别为1110、0100、0110、1111，试写出这4个中断源的中断处理次序(按降序排列)。若4个中断源同时有中断请求，画出CPU执行程序的轨迹。

9-18　现有A、B、C、D 4个中断源，其优先级由高向低按A→B→C→D顺序排列。若中断服务程序的执行时间为20μs，根据图9-30所示时间轴给出的中断源请求中断的时刻，画出CPU执行程序的轨迹。

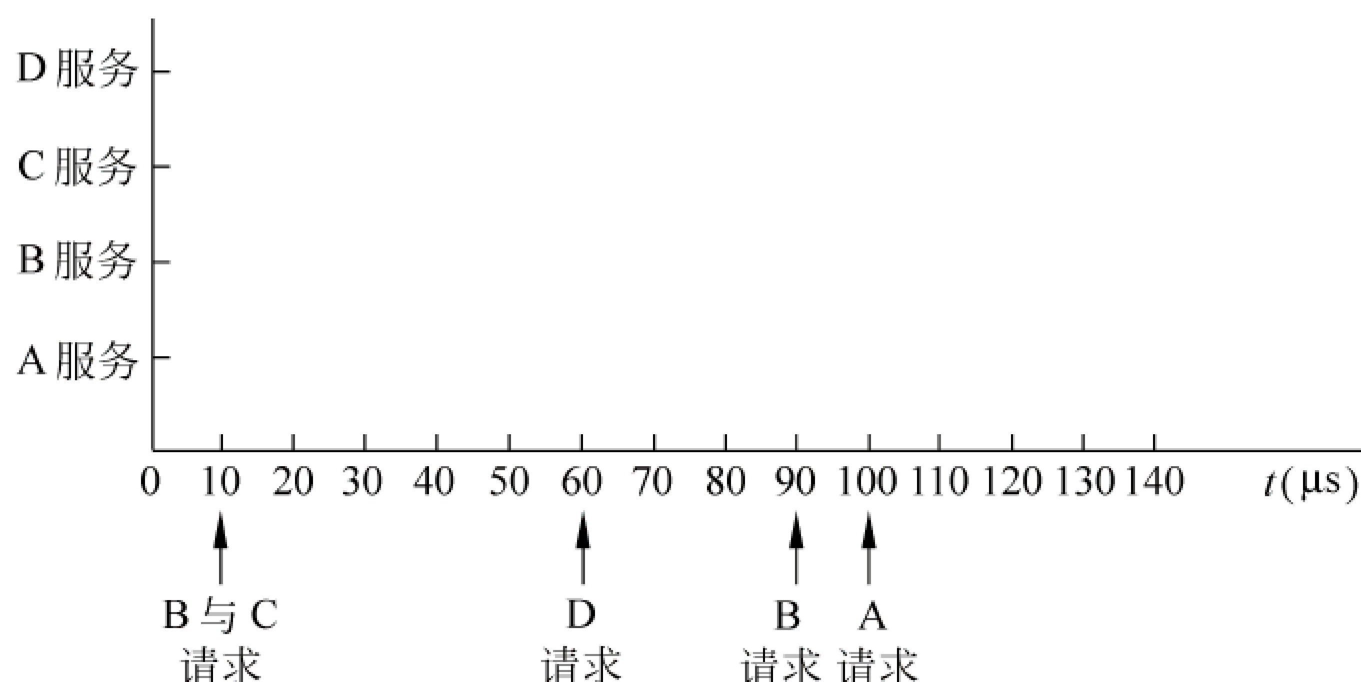


图9-30　中断请求时间轴

9-19　设某计算机有5级中断：L_0、L_1、L_2、L_3、L_4，其中断响应优先次序为：L_0最高，L_1次之，……，L_4最低。现在要求将中断处理次序改为L_1→L_3→L_0→L_4→L_2，试问：

(1) 各级中断服务程序中的各中断屏蔽码应如何设置(设每级对应一位，当该位为“0”，表示中断允许；当该位为“1”，表示中断屏蔽)？

(2) 若这5级同时都发出中断请求，试画出进入各级中断处理过程示意图。

9-20　实现多重中断应具备何种条件？如有A、B、C、D 4级中断，A的优先级最高，B次之，C再次之，D最低。如果在程序执行过程中，C和D同时申请中断，该先响应哪级中断？如果正在处理该中断时，A

和B又同时有中断请求,试画出该多级中断处理的流程来。

9-21 CPU响应DMA请求和响应中断请求有什么区别?为什么通常使DMA请求的优先级高于中断请求?

9-22 什么是DMA传送方式?试比较常用的3种DMA传送方法的优缺点?

9-23 实现DMA传送需要哪些硬件支持?

9-24 简述DMA传送的工作过程。

9-25 在主存接收从磁盘送来的一批信息时:

(1) 假定主存的周期为1μs,若采用程序查询方式传送,试估算在磁盘上相邻两数据字间必须具有的最短允许时间间隔是多少?

(2) 若改为中断方式传送,这个时间又会怎样?是否还有更好的传送方式?

(3) 在采用更好的传送方式下,假设磁盘上两数据字间的间隔为1μs,主存又要被CPU占有一半周期时间,试计算这种情况下主存周期最少应是多少。

9-26 磁盘机采用DMA方式与主机通信,若主存周期为1μs,能否满足传输速率为1MB/s的磁盘机的要求?此时CPU处于什么状态?若要求主存有一半时间允许CPU访问,该如何处理?

9-27 假定一个字长为32位的CPU的主频为500MHz,硬盘的传输速率为4MB/s。

(1) 采用中断方式进行数据传送,每次中断传输4字块数据。每次中断的开销(包括中断响应和中断处理的时间)是500个时钟周期,问CPU用于磁盘数据传送的时间占整个CPU时间的百分比是多少?

(2) 采用DMA方式进行数据传送,每次DMA传输的数据量为8KB。如果CPU在DMA预处理时花了1000个时钟周期,在DMA后处理时花了500个时钟周期,问CPU用于磁盘数据传送的时间占整个CPU时间的百分比为多少?

9-28 通道有哪些基本类型?各有何特点?

9-29 已知一个32位大型计算机系统具有两个选择通道和一个字节多路通道。每个选择通道连接两台磁盘机和两台磁带机,字节多路通道连接两台打印机、两台卡片输入机和10台CRT显示终端。假设这些设备的传输速率分别为:

磁盘机	800KB/s
磁带机	200KB/s
打印机	6.6KB/s
卡片输入机	1.2KB/s
CRT显示终端	1KB/s

求该计算机系统的最大I/O传输速率。

9-30 某计算机I/O系统中,接有一个字节多路通道和一个选择通道。字节多路通道包括3个子通道。其中,0号子通道上接有两台打印机(传输率为5KB/s);1号子通道上接有3台卡片输入机(传输率为1.5KB/s);2号子通道上接8台显示器(传输率为1KB/s)。选择通道上接两台磁盘机(传输率为800KB/s);5台磁带机(传输率为250KB/s),求I/O系统的实际最大流量。若I/O系统的极限容量为822KB/s,问能否满足所连接设备流量的要求?

9-31 试概括通道控制方式和DMA方式的异同点。

9-32 什么是通道指令?通道指令的结构如何?它与CPU指令有何区别?它们的执行过程相同吗?

9-33 简述通道操作的基本过程。

9-34 在通道控制方式下,I/O操作由通道控制,以达到CPU和I/O设备的并行操作,试问:

(1) 当通道正在进行I/O操作时,CPU能否响应其他中断请求?

(2) 若CPU能响应其他中断请求,是否会影响正在进行的I/O操作?

索　引

A～B

编译程序　compiler　11
八进制　octal　15
补码　two's complement　16
编址方式　addressed mode　54
变址寻址　indexed addressing　58
并行加法器　parallel adder　80
不恢复余数　non-restoring remainder　100
不归零制　non-return to zero(NRZ)　235
薄膜晶体管　thin film transistor(TFT)　279
并行接口　parallel interface　288

C

超大规模集成电路　very large scale integration(VLSI)　1
存储程序计算机　stored program computer　2
存储器　memory　2
长浮点数　long floating point number　26
操作码　operation code　49
程序计数器　program counter(PC)　50
存储器堆栈　stack storage　64
超长指令字　very long instruction word(VLIW)　74
串行加法器　serial adder　80
成组先行进位　block carry look ahead(BCLA)　83
存储层次　memory hierarchy　125
存储体　memory bank(MB)　126
存储单元　memory location　127
存储容量　memory size　128
存取速度　memory speed　128
存取周期　memory cycle　128
存取时间　memory access time　128
存储矩阵　memory matrix　133
存储器地址寄存器　memory address register(MAR)　145
存储器数据寄存器　memory data register(MDR)　145
错误校验与校正　error checking and correcting(ECC)　146
程序状态字寄存器　program status word register(PSWR)　170
垂直型微指令　vertical microinstruction　191
超线程　hyper-threading(HT)　210
磁道　track　234

磁盘地址　disk address　240
触摸板　touchpad　263
垂直扫描频率　vertical scan frequency　272
超级偏转向列　super twist nematic(STN)　279

D

大规模集成电路　large scale integration(LSI)　1
地址总线　address bus(AB)　5
带符号数　signed number　15
定点表示法　fixed-point representation　20
短浮点数　short floating point number　26
地址码　address code　49
堆栈　stack　63
多媒体扩展　multi media extension(MMX)　73
低位来的进位　carry in　79
地址译码　address decoding　126
大端　big-endian　127
动态随机存取存储器　dynamic random access memory(DRAM)　130
电可擦除可编程只读存储器　electrically erasable programmable read only memory(EEPROM)　138
段式虚拟存储器　segment virtual memory　164
段表　segment table　164
段页式虚拟存储器　segment page virtual memory　165
多线程并行　thread level parallelism(TLP)　212
单芯片多处理器　chip multiprocessor(CMP)　212
道密度　track density　234
电荷耦合装置　charge coupled device(CCD)　264
点阵打印机　dot-matrix printer　266
点距　dot pitch　272
端口　port　287
端口地址　port address　289

E

二进制　binary　3
二进制编码的十进制数　binary-code decimal(BCD)　34

F

冯・诺依曼型计算机　von neumann machine　3
辅助存储器　auxiliary storage　4
反码　one's complement　16
浮点表示法　floating-point representation　22
返回地址　return address　68
返回指令　return instruction　69

复杂指令系统计算机　complex instruction set computer(CISC)　74
分散控制　separate of control　181
分区域记录　zoned recording　244
非向量中断　non-vectored interrupt　297

G

高速缓冲存储器　cache memory　4
固件　firmware　9
规格化浮点数　normalized floating point number　23
归零制　return to zero(RZ)　235
改进的调频制　modified frequency modulation(MFM)　236
固态硬盘　solid state disk　257
轨迹球　trackball　263
隔行扫描　interlaced scanning　273
光栅　raster　273
关中断　interrupt disable　304

H

汇编程序　assembler　10
汉明校验码　hamming code　42
后进先出　last in first out(LIFO)　63
恢复余数　restoring remainder　99
行地址选通　row address select(RAS)　135
恒定线速度　constant linear velocity(CLV)　251
恒定角速度　constant angular velocity(CAV)　251

I～J

极大规模集成电路　extremely large scale integration(ELSI)　2
计算机系统　computer system　5
金属氧化物半导体　metal-oxide-semiconductor(MOS)　7
晶体管-晶体管逻辑　transistor-transistor logic(TTL)　7
兼容性　compatibility　9
解释　interpreter　10
阶码　exponent　22
基数　radix　22
奇校验　odd parity　40
寄存器寻址　register addressing　56
绝对寻址　absolute addressing　57
间接寻址　indirect addressing　57
寄存器间接寻址　register-indirect addressing　58
基址寻址　base addressing　59
寄存器堆栈　stack register　63

精简指令系统计算机　reduced instruction set computer(RISC)　74
加法器　adder　79
进位位　carry bit　80
记忆单元　cell　130
静态随机存取存储器　static random access memory(SRAM)　130
近期最少使用　least recently used(LRU)　161
机器周期　machine cycle　178
集中控制　centralization of control　181
键盘　keyboard　258
接触式图像感应装置　contact image sensor(CIS)　264
激光打印机　laser printer　270

K

控制器,控制单元　control unit(CU)　2
控制总线　control bus(CB)　5
扩展操作码　expanding operation code　53
可编程只读存储器　programmable read only memory(PROM)　138
可擦除可编程只读存储器　erasable programmable read only memory(EPROM)　138
快速页模式随机存储器　fast page mode dram(FPM DRAM)　151
扩展数据输出随机存储器　extended data output dram(EDO DRAM)　151
快速通道互联　quick path interconnect (QPI)　173
控制存储器　control memory(CM)　188
可写控制存储器　writeable control storage(WCS)　193
开中断　interrupt enable　300

L

累加寄存器　accumulator　4
立即寻址　immediate addressing　56
逻辑地址　logical address　59
逻辑移位指令　logical shift instruction　67
流式SIMD扩展　streaming SIMD extension(SSE)　73
列地址选通　column address select(CAS)　135
流水线　pipeline　202
廉价冗余磁盘阵列　redundant array of inexpensive disks(RAID)　245

M

每条指令时钟周期数　cycles per instruction(CPI)　13
每秒百万条指令　million instructions per second(MIPS)　14
每秒百万次浮点运算　million floating-point operations per second(MFLOPS)　14
模 module　17
每英寸磁道数　tracks per inch(TPI)　234
每毫米磁道数　tracks per millimeter(TPM)　234

每英寸的位数 bits per inch(BPI) 234
每毫米的位数 bits per millimeter(BPM) 234
每英寸点数 dots per inch(DPI) 265
每秒字符数 characters per second(CPS) 266
每分钟行数 lines per minute(LPM) 266
每分钟页数 pages per minute(PPM) 266

N～O

偶校验 even parity 40

P

平均故障间隔时间 mean time between failures(MTBF) 129
平均存取时间 average access time 234
喷墨打印机 inkjet printer 269

Q

全加器 full adder 79
全相联映像 full associate mapping 160
前端总线 front side bus 173
取指周期 instruction fetch cycle 178
全速命令排队 native command queuing(NCQ) 245

R

软件 software 8
热设计功耗 thermal design powerTDP 175
软错误 soft error 235

S

输入设备 input devices 2
输出设备 output devices 2
算术逻辑运算部件 arithmetic and logical unit(ALU) 4
数据总线 data bus(DB) 5
数据通路 data path 12
十进制 decimal 15
十六进制 hexadecimal 15
双字 doubleword 38
四字 quadword 38
双四字 double-quad word 38
数据传送指令 data movement instruction 65
输入输出指令 input/output instruction 69
随机存取存储器 random access memory(RAM) 124
顺序存取存储器 sequential access memory(SAM) 124

刷新　memory refresh　133
刷新周期　refresh cycle　133
死区　dead space　133
闪速存储器　flash memory　139
双数据传输率同步动态随机存储器　double data rate SDRAM(DDR SDRAM)　152
双通道内存技术　dual ddr　155
随机　random　161
数据流　data stream　169
水平型微指令　horizontal microinstruction　191
数据相关　data dependence　207
三态门　tri-state device　217
扇区　sector　240
鼠标器　mouse　262
水平扫描频率　horizontal scan frequency　272
扫描线　scan lines　273
随机存取内存数字模拟转换器　random access memory digital analog convertor(RAMDAC)　283
输入输出接口　input/output interface　286

T

特大规模集成电路　ultra large scale integration(ULSI)　2
通道　channel　5
统一代码　unicode　33
条件转移　jump if not　68
同步动态随机存储器　synchronous dram(SDRAM)　152
同步控制方式　synchronous control mode　181
同步通信　synchronous communication　223
通用异步收发器　universal asynchronous receiver transmitter(UART)　227
调相制　phase encoding(PE)　236
调频制　frequency modulation(FM)　236
图形显示　graphics display　271
通道程序　channel program　291
通道控制字　channel control word(CCW)　317
通道地址字　channel address word(CAW)　317
通道状态字　channel status word(CSW)　317
通道指令　channel instruction　317

U～W

外部设备　peripheral　3
位　bit　12
无符号数　unsigned number　15
尾数　mantissa　22
位移量　displacement　59

物理地址　physical address space　59
微操作　microoperation　177
微程序设计　microprogram design　184
微指令　microinstruction　185
微指令寄存器　microinstruction register　188
微程序计数器　microprogram counter　190
位密度　bit density　234
无源阵列　passive matrix　279

X

小规模集成电路　small scale integration(SSI)　1
虚拟机器　virtual computer　11
小数　fraction　20
循环冗余校验　cyclic redundancy check(CRC)　45
寻址方式　addressing mode　54
相对寻址　opposite addressing　59
先进后出　first in last out(FILO)　63
显式并行指令代码　explicit parallel instruction code(EPIC)　75
向高位的进位　carry out　79
先行进位　carry look ahead(CLA)　82
小端　little-endian　127
先进先出　first in first out(FIFO)　161
写直达法　write through　161
写回法　write back　162
虚拟存储器　virtual memory　163
虚地址　virtual address　163
寻道　track seeking　240
像素　pixel　265
向量中断　vectored interrupt　297
向量地址　vector address　297

Y

硬件　hardware　2
运算器　arithmetic unit　2
运算速度　arithmetic speed　13
原码　sign magnitude　16
移码　bias　24
隐含位　hidden bit　26
有效地址　effective address(EA)　56
页面寻址　page addressing　60
运算指令　arithmetic instruction　66
移位指令　shift instruction　67

溢出　overflow　86
移位器　shifter　90
掩膜 ROM mask_programmable ROM(MROM)　138
页式虚拟存储器　page virtual memory　164
页表　page table　164
异步控制方式　asynchronous control mode　181
异步通信　asynchronous communication　224
硬错误　hard error　235
游程长度受限码　run length limited(RLL)　237
硬盘存储器　hard disk storage　239
硬盘控制器　hard disk controller(HDC)　239
圆柱面　cylinder　240
一次写入多次读出　write one read many(WORM)　248
阴极射线管　cathode ray tube(CRT)　271
液晶显示器　liquid crystal display(LCD)　271
有源阵列　active matrix　279

Z

中规模集成电路　medium scale integration(MSI)　1
中央处理器　central processing unit(CPU)　3
主机　mainframe　3
总线　bus　5
字节　byte　12
字　word　12
整数　integer　21
指令格式　instruction format　49
指令系统　instruction set　49
指令寄存器　instruction register　56
直接寻址　direct addressing　57
栈顶　stack head　63
栈底　stack bottom　64
栈指针　stack pointer　64
转移指令　jump instruction　68
转移条件　jump condition　68
子程序调用指令　subroutine call instruction　68
子程序入口地址　subroutine entry address　68
主程序　main routine　68
子程序　subroutine　68
子程序嵌套　subroutine nesting　68
子程序递归　subroutine recursive　68
只读存储器　read-only memory(ROM)　124
直接存取存储器　direct access memory(DAM)　124
主存带宽　memory bandwidth　129

再生　regeneration　132
直接映像　direct mapping　160
组相联映像　set associate mapping　161
指令流　instruction stream　169
直接媒体接口　direct media interface (DMI)　173
指令译码器　instruction decoder　176
指令周期　instruction cycle　178
执行周期　execution cycle　178
中断周期　interrupt cycle　178
指令级并行　instruction level parallelism(ILP)　212
总线仲裁　bus arbitration　221
总线控制器　bus controller　221
终端　terminal　231
字符显示　character display　271
直接存储器存取　direct memory access(DMA)　291
中断服务程序　interrupt service routine　295
中断源　interruption source　297
中断请求　interruption request　297
中断判优　interrupt arbiter　298
中断响应　interrupt response　300
中断允许触发器　interrupt-enable flip-flop　300
中断处理　interruption handing　300
中断现场　interrupt spot　302
中断嵌套　interrupt nesting　303
中断屏蔽　interruption mask　303
中断屏蔽寄存器　interrupt mask register　304
中断屏蔽字　interrupt mask word　304
总线请求　bus request　310
总线响应　bus respond　310
周期挪用　cycle steal　313

IEEE(电气和电子工程师学会)　Institute of Electrical and Electronic Engineers　26
ASCII(美国信息交换标准字符码)　American Standard Code for Information Interchange　29
IA(Intel 结构)　Intel Architecture　38
Booth 算法　Booth's algorithm　94
DIP(双列直插封装)　dual in-line package　139
SIMM(单列直插存储模块)　single in-line memory module　140
DIMM(双列直插存储模块)　dual in-line memory module　140
RIMM(rambus 直插存储模块)　rambus in-line memory　140
SO-DIMM(小外形双列直插存储模块)　small outline dual in-line memory module　140
RDRAM(rambus DRAM)　154
ISA(工业标准结构)　industry standard architecture　225
MCA(微通道结构)　micro channel architecture　225

EISA(扩展工业标准结构) extended industry standard architecture 225
VESA 局部总线 video electronics standard association local bus 225
PCI(周边元件扩展接口) peripheral component interconnect 225
AGP(图形加速端口) accelerated graphics port 225
USB(通用串行总线) universal serial bus 228
eSATA(外部串行 ATA) external serial ATA 229
IDE(集成驱动器电路) integrated drive electronics 242
SATA(串行 ATA) serial ATA 243
SCSI(小型计算机系统接口) small computer system interface 243
SAS(串行连接 SCSI) serial attached SCSI 243
CD-ROM(只读光盘) compact disc read only memory 248
CD-R 光盘 compact disc-recordable 248
CD-RW 光盘 compact disc-rewritable 248
DVD 光盘 digital versatile disc 248
BD 光盘(蓝光盘) blu-ray disk 248
U 盘(USB 闪存驱动器) USB flash disk 257
A/N(字母数字)模式 alpha number mode 273
APA 模式 all-points addressable mode 274
MDA(单色显示适配器) monochrome display adapter 281
CGA(彩色图形适配器) color graphics adapter 281
EGA(增强的图形适配器) enhanced graphics adapter 281
VGA(视频图形阵列) video graphics array 281
XGA(扩展图形显示) extended graphics array 282

参 考 文 献

[1] 蒋本珊.计算机组成原理.3版.北京：清华大学出版社,2014.
[2] 唐朔飞.计算机组成原理.2版.北京：高等教育出版社,2008.
[3] 王爱英.计算机组成与结构.3版.北京：清华大学出版社,2001.
[4] Scott Mueller. PC硬件工程师手册.吕俊辉,等,译.北京：机械工业出版社,2002.
[5] Patterson，Hennessy.计算机组成和设计硬件/软件接口.2版.郑纬民,译.北京：清华大学出版社,2003.
[6] 李亚民.计算机组成与系统结构.北京：清华大学出版社,2000.
[7] William Stallings. computer organization and architecture(影印版).北京：高等教育出版社,2001.



“十二五”普通高等教育本科国家级规划教材
21世纪大学本科计算机专业系列教材

近期出版书目

- 计算概论(第 2 版)
- 计算概论——程序设计阅读题解
- 计算机导论(第 4 版)
- 计算机导论教学指导与习题解答
- 计算机伦理学
- 程序设计导引及在线实践(第 2 版)
- 程序设计基础(第 2 版)
- 程序设计基础习题解析与实验指导
- 程序设计基础(C 语言)(第 2 版)
- 程序设计基础(C 语言)实验指导(第 2 版)
- 离散数学(第 3 版)
- 离散数学习题解答与学习指导(第 3 版)
- 数据结构(STL 框架)
- 算法设计与分析(第 2 版)
- 算法设计与分析习题解答与学习指导(第 2 版)
- 算法设计与分析(第 4 版)
- 算法设计与分析习题解答(第 4 版)
- C ++ 程序设计(第 3 版)
- Java 程序设计(第 2 版)
- 面向对象程序设计(第 3 版)
- 形式语言与自动机理论(第 3 版)
- 形式语言与自动机理论教学参考书(第 3 版)
- 数字电子技术基础
- 数字逻辑
- FPGA 数字逻辑设计
- 计算机组成原理(第 4 版)
- 计算机组成原理教师用书(第 4 版)
- 计算机组成原理学习指导与习题解析(第 4 版)
- 微机原理与接口技术(第 2 版)
- 微型计算机系统与接口(第 2 版)
- 计算机组成与系统结构(第 2 版)
- 计算机组成与体系结构习题解答与教学指导(第 2 版)
- 计算机组成与体系结构(第 2 版)
- 计算机系统结构教程
- 计算机系统结构学习指导与题解
- 计算机系统结构实践教程
- 计算机操作系统(第 2 版)
- 计算机操作系统学习指导与习题解答
- 编译原理
- 软件工程(第 3 版)
- 计算机网络(第 4 版)
- 计算机网络教师用书(第 4 版)
- 计算机网络实验指导书(第 3 版)
- 计算机网络习题解析与同步练习(第 2 版)
- 计算机网络软件编程指导书
- 人工智能
- 多媒体技术原理及应用(第 2 版)
- 计算机图形学
- 计算机网络工程(第 2 版)
- 计算机网络工程实验教程
- 信息安全原理及应用